Getting Started With Kubernetes

Getting Started With Kubernetes

The Book is primarily intended for Beginners who would like to try their hands on the orchestration engine “kubernetes”. When i first started learning kubernetes, i started with the documentation but it is always complex. I had to google a lot for understanding the components. The hard part is understanding the kubernetes architecture.

The Books talks from basics of kubernetes to the deep parts of deployment strategies.
I tried to keep the content as simple as possible.

This Book assumes that you have some knowledge on Container runtime like Docker, Rkt or CTI-O. We will be using CentOs 7 and Docker are some of our components.

The code that is used in this book is available at here

Introduction - What is kubernetes

We all know how containers are helping the world with their small sizes , easy to scale and run any type of applications. The major problem with the containers are management and maintenance. How can we have same application, running in multiple containers are managed?. How can we have an application container started automatically when existing container goes down?.

The answer is kubernetes. Kubernetes is a open source orchestration engine for deployment, scaling , maintenance and management of containers or containerized applications. Kubernetes built by google provides mechanism for application deployment, scheduling , updating, maintenance and scaling. One major advantage of the kubernetes is that it makes sure to manage the containers to ensure that state of the cluster matches the user intentions.

So what does a kubernetes do?

The main purpose is to make it easier to organize and schedule your application across a fleet of machines. At a high level we can think of this as an operating system for your cluster. Basically it allows you not to worry about what type of specific machine in your datacenter your application container is running. It also provides health check facilities and scaling your application containers based on load.

Kubernetes provides much of the same functionality as Infrastructure as a Service APIs, but aimed at dynamically scheduled containers rather than virtual machines, and as Platform as a Service systems, but with greater flexibility, including:
mounting storage systems
distributing secrets
application health checking
replicating application instances
horizontal auto-scaling
naming and discovery
load balancing
rolling updates
resource monitoring
log access and ingestion
support for introspection and debugging, and identity and authorization.

Basics

K8 is an open source container management and orchestration tool based on go Language. This is lightweight and portable application.

Declarative vs imperative configuration
Many systems that are build now are based on declarative configuration. This is one of the main thing behind k8. What does this mean?

A declarative configuration defines how a system should be and the system will take care of that. If the user says "i want to have five replicas of my web container running all times". The system in turn will not understand these hence we define them in a Json or a Yml format. K8 will then read this statement and will make sure to have five replicas of the web container running all the time. This also means that if 1 of the 5 is down, k8 will take care of starting that up again and makes sure that 5 are running all the time. This means the system will do the self-correcting and healing also.

In the Imperative configuration , the user will perform the actions which means starting the five replicas of the web container. These are plan "run as" statements.

Controllers - Controllers are another important back bone of k8. K8 is not based on central system in turn is works as decentralized. This means components in k8 work independently. Controllers or Reconciliation loops does the job for us. K8 is based on large number of controller each performing its own job. A Controller can be taught as a control loop that watches the state and make sure it is always in desired state. If not change it to desired state. Each controller will have its own job and is unaware of the other controllers and their role.Some of these controllers include replication ,endpoint, namespace etc etc.

All these controllers will be managed by kubernetes controller manager which is a daemon that manages all these controllers.

Labeling - Labeling is another important back-bone of the K8 cluster. Take the above case as defining "i want to have five replicas of my web container running all times" , now how does k8 make sure that 5 replicas are running. How can it identify that one of the web replica is down. These identifications are done by using the Labels. Each component will be identified with the label that is assigned to that. K8 follows the labeling model in a different way. Rather than saying "these are part of my label group" it says "label group can container these".

In the second case ,we can add any number of components and still assign them to a group making the label dynamic with multiple components. These labels are defined in a key:value pair and are queried using the label queries or selectors.

Along with labels we have annotations. Label are defined to identify the component and annotations are defined with general metadata that cannot be quieres. The meta data can be some time like an icon to display etc.

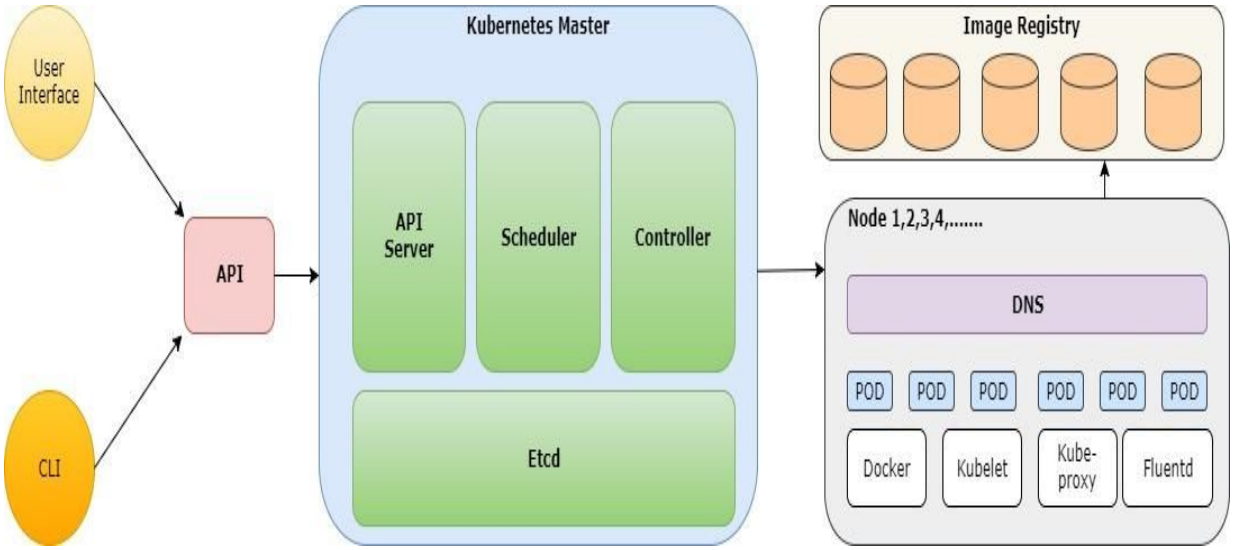
Kubernetes Components

As we already know k8 is a combination of multiple components. The description of the Kubernetes components below breaks them into these three groupings. The components that run on master nodes, the components that run on all nodes and the components that run scheduled onto the cluster.

The major components include,

Master Node or control plane - Etcd , Api Server , Scheduler , Container runtime and Controller
Worker Node - Pods , Kubelet , Kube-proxy , Container runtime and CNI Implemented Network like flannel,Weave etc, Image Registry
AddOns Components

The component architecture of kubernetes looks,



With multiple components in K8, it can be hard on how they talk to each other. In this article we will see what each component does in detail and also see how they talk to each other.

Every Component in K8 talks to Api Server. No Component talks to the Etcd other than Api Server. All Communication from Control Plane to worker nodes will happen only from Api Server. The communication between each component happens by a Rest based calls.

Before going in deep understanding of the components, let's get the status of the components running,

```
[root@manja17-I13330 kubenetes-config]# kubectl get po -o
custom-columns=POD:metadata.name,NODE:spec.nodeName --sort-by spec.nodeName -n
kube-system
POD                                     NODE
kubernetes-dashboard-7d5dcdb6d9-s967p   manja17-i13330
weave-net-rz5bh                         manja17-i13330
kube-apiserver-manja17-i13330           manja17-i13330
kube-controller-manager-manja17-i13330  manja17-i13330
kube-scheduler-manja17-i13330           manja17-i13330
kube-proxy-dcnmw                        manja17-i13330
etcd-manja17-i13330                     manja17-i13330
kube-proxy-js69w                        manja17-i14021
weave-net-255pb                         manja17-i14021
kube-proxy-ww4s5                        manja17-i14022
kube-dns-86f4d74b45-fvrtb               manja17-i14022
heapster-5b748fbdc5-cxtsq               manja17-i14022
weave-net-w582l                         manja17-i14022
```

Note - All the components of K8 run under the Kube-system name space. In the above Manja17-i13330 is master
Manja17-i14021 and manja17-i14022 are nodes

Let’s see the status of the components using,

```
[root@manja17-I13330 ~]# kubectl get componentstatuses
NAME                 STATUS    MESSAGE             ERROR
scheduler            Healthy   ok
controller-manager   Healthy   ok
etcd-0               Healthy   {"health": "true"}
```

Let’s start digging the Components,

Control Plane or Master Node Components - ETCD

The major k8 components run on the master nodes. There will be very limited number of these nodes based on our cluster. These master nodes run the important components of K8. These masters nodes will always be in odd numbers like 1,3 or 5 etc. This is because of the Quorum. A Quorum is not necessarily a majority of members of the group but the minimum needed in order to conduct business. For example, of 2 members of the group are absent, there can still be a quorum meaning they can conduct business without them.

If you have 3 masters and 2 of them are out of business , we can still run the cluster with one master.

Etcd - This is the very core component and heart of the cluster. This just stores data as a key value pair. Etcd is an open-source distributed key-value store that serves as a backbone for cluster coordination and state management.

For example , if we have a job scheduler that need to notify a machine that has job to do and once the job is done it again needs to share this information with other systems. Now we will need source of truth box to save the details so that it is sent to other components on a timely and reliable manner. This is where ETCD comes into picture. It saves the job details so that a coordination engine will read and pass the data to other components. So anything happening in the cluster will be saved to the ETCD.

Some of the main features of Etcd are,
Compare-and-swap (CAS) - This is one of the feature of Etcd. This is an atomic operation that result in comparing the contents of the location with a given value and only if they are same modifies the contents with new value. In Etcd if we have written a key:value , and if we want to save new value, it uses the same concept of Compare-and-swap. These assurances enable the system to safely have multiple threads manipulating data in etcd without the need for pessimistic locks which can significantly reduce throughput to the server.

Watches - Etcd also has a Watch protocol which watches for key changes. This watch works in a different way, rather than client polling for changes to the keys watch waits for changes to keys by continuously watching them and sending the details back to the client

RAFT - RAFT algorithm which make sure that even if one of the etcd goes down or fails there will be enough replication available to store data and once the dead one is available it re connects to the cluster.

How Does K8 Use ETCD?
All K8 objects like pods, deployments, service etc that are created in a cluster are stored in Etcd so that they survive failures. K8 allows to run multiple instances of etcd.The Only component that talks to etcd is Api Server. All other components read and write data to the etcd using the Api server.

Etcd implements hierarchical structure which makes key-value pairs similar to files in the file system. Each key in etcd is either a directory, which contains other keys, or is a regular key with a corresponding value. Depending on the value being stored (if slash includes), the value can be located in multiple sub-directories.

Control Plane or Master Node Components - API Server

Since we know that etcd stores data and for having consistent data we should not be allowing multiple things to modify data. This is where Api server comes into picture.

The job of the api server is to mediate the communication between etcd and other clients. The Api server is the only one who have access to the Etcd server. When a client makes a call, he does that to Api server which in turn connects to the etcd server to obtain data. Api server provides CRUD interface for querying and modifying state over a RESTful API calls.

Besides Storing data,it also performs validation of those objects like pod manifest etc. One another things that the API server does is to handle the locking, so that call to change an objects from the API server are handled correctly and never overridden by other clients in the event of concurrent update.

Here are the steps as what happens inside the API Server,

1. Authentication - Once the client request for creating a resource (pod, Deployment) comes to API , authentication is done to identify the client. Multiple authentication plugins are configured in the Api server which will read the request one after the another until one identifies the client sending the request. Since the request are in Rest format which is HTTP request, Api server will read the headers and identify the client by extracting the user name , user id, group id etc.
2. Authorization - Once the authentication is done, authorization comes into picture. More than one authorization plugins are configured to determine that the authenticated user can perform the action suggested in the request.
   For example, if a request for pod creation comes these plugin checks if that user has Permissions to create pod , in what namespace etc.
3. Admission Control Plugins - In the 3 stage admission control plugins comes into into picture. The Job of these is intercept the request before the Api server making that persistent. It intercepts the request and validates if anything is missing, setting default values. For example, let's say if we have a Namespace which has resource quota set and if we are creating a pod in this namespace , these controls will set the resource limits for the pod based on the namespace.
4. The request is then stored in the etcd.

One important thing that we need to know is that Api server gets the request from client and once all above steps are done it will save to the etcd server. It will full-fill any requests like , if Api server receives a request for pod creation it performs all the above steps and then store in the Etcd. It will not create the pod or tell to any other component to create the pod.

One another important job that the Api server does is sending notifications to the clients. For example , if a client request for creation of the pod, the data is saved into the etcd and a notification is sent to the interested parties. These interested parties can either create the

pod or perform some other action based on the notification sent by the APi server. These interested parties can request for the changes from Api server that they are interested in.

Clients or interested parties open a Http connection to the Api Server and using this connection the client will receive notification (like modification , creation) to the resources that are interested in or requested for.

Kubectl is one of the client for Api server.Using the command "kubectl get pods --watch", we don't need to poll for the list of pods in turn Api server will send the details to the kubectl which is watching.

Exploring the Api Interface - Api is a library of functions and procedures whose interfaces are exposed so that external applications can use them. If we need to develop any scripts we need to have an understanding of the functionality that is exposed by Kubernetes.
In many cases we will use the kubectl command to talk with the api server in performing the tasks. K8 Api is based on Rest implementation which means we can make a Rest call to the api server using linux command line tools like curl or any other tool that can make a rest call.

Api provides Url for endpoints and these endpoints provide access to specific functionality. Let use the kubectl command to access Api. for this we will run the "kubectl proxy" command from where we want to access the Api.

```
[root@manja17-I13330 ~]# kubectl proxy --port=8001 &
[1] 20120
[root@manja17-I13330 ~]# Starting to serve on 127.0.0.1:8001
```

The above command starts a proxy to the Kubernetes API server.Now use the curl command to access the Api as,

```
[root@manja17-I13330 ~]# curl http://localhost:8001/
{
  "paths": [
    "/api",
    "/api/v1",
    "/apis",
    "/apis/",
    "/apis/admissionregistration.k8s.io",
    "/apis/admissionregistration.k8s.io/v1beta1",
    "/apis/apiextensions.k8s.io",
    "/apis/apiextensions.k8s.io/v1beta1",
    "/apis/apiregistration.k8s.io",
    "/apis/apiregistration.k8s.io/v1",
    "/apis/apiregistration.k8s.io/v1beta1",
    "/apis/apps",
    "/apis/apps/v1",
    "/apis/apps/v1beta1",
    "/apis/apps/v1beta2",
    "/apis/authentication.k8s.io",
    "/apis/authentication.k8s.io/v1",
```

"/apis/authentication.k8s.io/v1beta1",
"/apis/authorization.k8s.io",
"/apis/authorization.k8s.io/v1",
"/apis/authorization.k8s.io/v1beta1",
"/apis/autoscaling",
"/apis/autoscaling/v1",
"/apis/autoscaling/v2beta1",
"/apis/batch",
"/apis/batch/v1",
"/apis/batch/v1beta1",
"/apis/certificates.k8s.io",
"/apis/certificates.k8s.io/v1beta1",
"/apis/events.k8s.io",
"/apis/events.k8s.io/v1beta1",
"/apis/extensions",
"/apis/extensions/v1beta1",
"/apis/networking.k8s.io",
"/apis/networking.k8s.io/v1",
"/apis/policy",
"/apis/policy/v1beta1",
"/apis/rbac.authorization.k8s.io",
"/apis/rbac.authorization.k8s.io/v1",
"/apis/rbac.authorization.k8s.io/v1beta1",
"/apis/storage.k8s.io",
"/apis/storage.k8s.io/v1",
"/apis/storage.k8s.io/v1beta1",
"/healthz",
"/healthz/autoregister-completion",
"/healthz/etcd",
"/healthz/ping",
"/healthz/poststarthook/apiservice-openapi-controller",
"/healthz/poststarthook/apiservice-registration-controller",
"/healthz/poststarthook/apiservice-status-available-controller",
"/healthz/poststarthook/bootstrap-controller",
"/healthz/poststarthook/ca-registration",
"/healthz/poststarthook/generic-apiserver-start-informers",
"/healthz/poststarthook/kube-apiserver-autoregistration",
"/healthz/poststarthook/rbac/bootstrap-roles",
"/healthz/poststarthook/start-apiextensions-controllers",
"/healthz/poststarthook/start-apiextensions-informers",
"/healthz/poststarthook/start-kube-aggregator-informers",
"/healthz/poststarthook/start-kube-apiserver-informers",
"/logs",
"/metrics",
"/openapi/v2",
"/swagger-2.0.0.json",
"/swagger-2.0.0.pb-v1",
"/swagger-2.0.0.pb-v1.gz",

```
    "/swagger.json",
    "/swaggerapi",
    "/version"
  ]
}
```

Each of the above line talks about the interface that is being exposed by the kubernetes. Let's check for a specific interface,

```
[root@manja17-I13330 ~]# curl http://localhost:8001/api/v1 | less
{
  "kind": "APIResourceList",
  "groupVersion": "v1",
  "resources": [
    {
      "name": "bindings",
      "singularName": "",
      "namespaced": true,
      "kind": "Binding",
      "verbs": [
        "create"
      ]
    },
*******
```

If we can see that this is a APIResourceList kind which provides various resource bindings. If we grep the above command more specifically we can see,

```
[root@manja17-I13330 ~]# curl http://localhost:8001/api/v1 | grep name | grep -v namespaced
      "name": "bindings",
      "name": "componentstatuses",
      "name": "configmaps",
      "name": "endpoints",
      "name": "events",
      "name": "limitranges",
      "name": "namespaces",
      "name": "namespaces/finalize",
      "name": "namespaces/status",
      "name": "nodes",
      "name": "nodes/proxy",
      "name": "nodes/status",
      "name": "persistentvolumeclaims",
      "name": "persistentvolumeclaims/status",
      "name": "persistentvolumes",
      "name": "persistentvolumes/status",
      "name": "pods",
      "name": "pods/attach",
      "name": "pods/binding",
      "name": "pods/eviction",
```

```
    "name": "pods/exec",
    "name": "pods/log",
    "name": "pods/portforward",
    "name": "pods/proxy",
    "name": "pods/status",
    "name": "podtemplates",
    "name": "replicationcontrollers",
    "name": "replicationcontrollers/scale",
    "name": "replicationcontrollers/status",
    "name": "resourcequotas",
    "name": "resourcequotas/status",
    "name": "secrets",
    "name": "serviceaccounts",
    "name": "services",
    "name": "services/proxy",
    "name": "services/status",
```

This is the same that we will be using when defining a manifest file. For example if we are creating a name space ,we define the manifest file as

```
[root@manja17-I13330 kubenetes-config]# cat basic-namespace.yml
apiVersion: v1
kind: Namespace
metadata:
  name: sample-testing
```

We can the first element that is being passed it the apiVersion: v1 which in turn uses these. This also talks about what operations that can be performed using the namespaces. If you hit the api server with above command and check for the namespace element, we see

```
 {
    "name": "namespaces",
    "singularName": "",
    "namespaced": false,
    "kind": "Namespace",
    "verbs": [
      "create",
      "delete",
      "get",
      "list",
      "patch",
      "update",
      "watch"
    ],
    "shortNames": [
      "ns"
    ]
  },
```

It gives all the operations that can be done on the namespaces. If you want to find out all the available namespaces currently exist we can use,

```
[root@manja17-I13330 kubenetes-config]# curl http://localhost:8001/api/v1/namespaces | grep name | grep -v "selfLink"
      "name": "default",
      "name": "kube-public",
      "name": "kube-system",
```

Using this "curl http://localhost:8001/api/v1/namespaces/default/pods" we can a list of all pods running in the default namespace

Now if we need to find more details about a pod running in a namespace we can use,

```
[root@manja17-I13330 kubenetes-config]# curl http://localhost:8001/api/v1/namespaces/default/pods/my-nginx
{
  "kind": "Pod",
  "apiVersion": "v1",
  "metadata": {
    "name": "my-nginx",
    "namespace": "default",
    "selfLink": "/api/v1/namespaces/default/pods/my-nginx",
    "uid": "e68afd34-8f0d-11e8-a791-020055e1ea1d",
    "resourceVersion": "7837400",
    "creationTimestamp": "2018-07-24T06:50:50Z",
    "labels": {
      "env": "dev"
    }
  },
  "spec": {
    "volumes": [
      {
        "name": "default-token-fx8mm",
        "secret": {
          "secretName": "default-token-fx8mm",
          "defaultMode": 420
        }
      }
    ],
    "containers": [
      {
        "name": "my-nginx",
        "image": "nginx",
        "ports": [
          {
            "containerPort": 80,
            "protocol": "TCP"
          }
        ],
```

```
      "resources": {

      },
      "volumeMounts": [
        {
          "name": "default-token-fx8mm",
          "readOnly": true,
          "mountPath": "/var/run/secrets/kubernetes.io/serviceaccount"
        }
      ],
      "terminationMessagePath": "/dev/termination-log",
      "terminationMessagePolicy": "File",
      "imagePullPolicy": "Always"
    }
  ],
  "restartPolicy": "Always",
  "terminationGracePeriodSeconds": 30,
  "dnsPolicy": "ClusterFirst",
  "serviceAccountName": "default",
  "serviceAccount": "default",
  "nodeName": "manja17-i14021",
  "securityContext": {

  },
  "schedulerName": "default-scheduler",
  "tolerations": [
    {
      "key": "node.kubernetes.io/not-ready",
      "operator": "Exists",
      "effect": "NoExecute",
      "tolerationSeconds": 300
    },
    {
      "key": "node.kubernetes.io/unreachable",
      "operator": "Exists",
      "effect": "NoExecute",
      "tolerationSeconds": 300
    }
  ]
},
"status": {
  "phase": "Running",
  "conditions": [
    {
      "type": "Initialized",
      "status": "True",
      "lastProbeTime": null,
      "lastTransitionTime": "2018-07-24T06:49:26Z"
```

```
      },
      {
        "type": "Ready",
        "status": "True",
        "lastProbeTime": null,
        "lastTransitionTime": "2018-07-24T06:49:36Z"
      },
      {
        "type": "PodScheduled",
        "status": "True",
        "lastProbeTime": null,
        "lastTransitionTime": "2018-07-24T06:50:50Z"
      }
    ],
    "hostIP": "10.131.36.181",
    "podIP": "10.38.0.3",
    "startTime": "2018-07-24T06:49:26Z",
    "containerStatuses": [
      {
        "name": "my-nginx",
        "state": {
          "running": {
            "startedAt": "2018-07-24T06:49:36Z"
          }
        },
        "lastState": {

        },
        "ready": true,
        "restartCount": 0,
        "image": "docker.io/nginx:latest",
        "imageID":
"docker-pullable://docker.io/nginx@sha256:4a5573037f358b6cdfa2f3e8a9c33a5cf11bcd167
5ca72ca76fbe5bd77d0d682",
        "containerID":
"docker://ac58e1679ffb9c74063431ae280fce7d35fa55e99883ef70c235d5e32b427416"
      }
    ],
    "qosClass": "BestEffort"
  }
}
```

Control Plane or Master Node Components - SCHEDULER

Now we have a component to save our data and another component which connects to the data server. Let's say that you are creating a Pod. You define a pod definition file and you make a call to create the pod. Now the api server will take the call and save the pod details into etcd and create the pod. Now the question is where does this pod gets created?. How does k8 know where to create the pod?

This is done by the scheduler. The Job of the scheduler is to scan for unscheduled pods and then determines which is the best worker node to run the pod.

How does this work?
Scheduler using the watch mechanism wait for newly created pods through the Api Server and assign a node to each new pod. The the scheduler identifies the correct node for hosting the pod and will update the API server with the details. The details of the node that the pod should run is updated in the etcd and kubectl on every node which is watching will get the details and kubectl will create the pod.

But how does scheduler identifies the node for hosting the pod?
Scheduler will run pre-defined functions to identify which node is suitable for hosting the pod. The functions can range from,

1. Is this Node running out of resources
2. Is the pod defined with this node for creation?
3. Does the node have any label that matches the node selector defined in the pod definition.
4. Do the pod needed port is available in here or not?

These are some of the functions that run and based on the results it identifies a suitable node for hosting the pod.

A default scheduler will run if none is specified. We can have a cluster with no scheduler specified which in turn lets us to do things manually.

Control Plane or Master Node Components - Controller

As we know that etcd stores data, Api server collects request and save them to etcd and scheduler identifies suitable nodes for the pods to host. But who will make sure to maintain the exact state. That is to make sure the number of pod replicas are running as defined in the pod manifest. This is making sure that a desired state is obtained. This is the place where controller or control manager come into picture. The control manager is combination of many controllers which perform their job as defined.

Some of the controller include,

- Replication Manager
- ReplicaSet, DaemonSet, and Job controllers
- Deployment controller
- Node controller
- Service controller
- Namespace controller
- PersistentVolume controller

These controller consist of a reconciliation loops which will make sure that a resource is in desired state. Some include,

Replication controller ensures all the replication controllers run on the pod desired number. A **ReplicationController** ensures that a specified number of pod replicas are running at any one time.

Node Controller - This controller responds when nodes go down. This will take care of node health status check. Based on the status it will make sure the node is removed if unreachable. This also make sure to keep all the nodes in the cluster up to date.

End Point Controller - This takes care of associating the relationship between services and pods.

Service Account and Token controller are used to control default account and API access tokens.

As defined controller perform the task they are defined for. First they watch for the API server to send updates on the resource changes. As said each controller contains a reconciliation loop which check for the desired state with the actual state. If different an appropriate controller will start and do the job of making the actual and desired state similar. These controller also watch for Api updates and act accordingly.

We understood what each of the component does and how they talk, but none of the control plane or master node components will create the pod. They talk between them and make a call to a process running on worker node to actually create the pod. This process is called as Kubelet

Work Node Components

Now once the components are ready on the Master node, we need to understand the components that run the worker nodes. The worker nodes are basically where our workload runs. These are the nodes where we run our application containers etc.

Kubelet - In simple terms , kubelet is a process that run on the every worker node of the cluster to perform jobs like creating pod etc. The first thing it does is to register the node it is running with the Api server. Then it continuously monitors the Api server for pods that have been scheduled on that node. One identified it tells the underlying container runtime like docker to start the containers. The Kubelet is also the component that runs the container liveness probes, restarting containers when the probes fail.

The Kubelet constantly monitors running containers and reports their status, events, and resource consumption to the API server. This is the component which will send all details regarding the node to Api server which intrun save them to the etcd. Checking these details ,the scheduler will identify other details and select the correct node for pods hosting.

Kube-Proxy - Kube-proxy is a network proxy and a load balancer which reflects kubernetes networking services on each node. Every worker node will have a kube-proxy whose purpose is to make sure clients can connect to the services we define using the Kubernetes Api. kube-proxy will take care of making sure the when someone hits the service IP with a port , it sends the request to the Pod that is actually backing the service.

Now once we create a pod, we need to have something to access the application running in that pod. Now this some thing is called services in k8. Every service will have a name and a end point (which is the original application point to connect). Kube-proxy will always watch for the endpoints for all services in the cluster. This will then re-program the network on its node, so that network requests to the ip address of the service will be re routed to the original endpoint.Every Service in Kubernetes gets a virtual IP address, the kube-proxy is the daemon responsible for defining and implementing the local load-balancer that routes traffic from Pods on the machine to Pods, anywhere in the cluster, that implement the Service.

It also does the job of load balancing if multiple pods are backing a service. When a client makes a call to a service, Iptables come in between intercept the request and forwards to the kube-proxy which in turn send that to the pod backing the service.

Kubernetes proxy can be used to access a pod locally without exposing that. Generally when you start the kube-proxy the service are accessible on the host where the kube-proxy is started.

Container Runtime - Container runtime is same for both master and worker nodes.Container runtime is a software responsible for running containers. K8 supports several runtime including docker,rkt,runc and any OCI runtime spec implementation

AddOn Components

Addons are pods and services that implement cluster features. The pods may be managed by Deployments, ReplicationControllers, and so on. Namespaced addon objects are created in the **kube-system** namespace.

Kube-dns

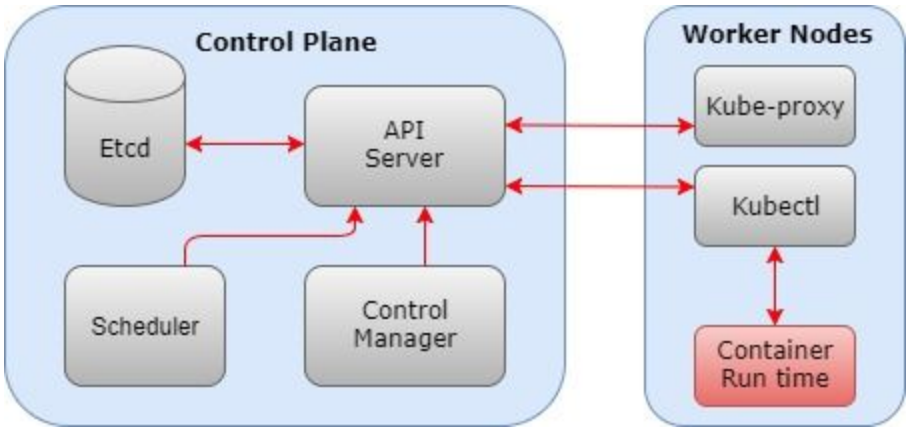
All Pods in the cluster when created will be configured with the cluster internal DNS server by default. All Pods are by default set to use the internal Dns server. This helps in finding the pods easily.

Kube-dns runs a pod in the kube-system and the ip address of this is added to /etc/resolv.conf of every pod that gets created. This kube-dns use the watch mechanism to observer any changes to the service or endpoints and updates it Dns records with every change.

```
[root@manja17-I13330 kubenetes-config]# kubectl get deploy --namespace=kube-system
NAME                DESIRED   CURRENT   UP-TO-DATE   AVAILABLE   AGE
kube-dns            1         1         1            1           72d
```

How does all these components talk to each other?

Now that we have seen how each component works, lets see how each of the control plane , work node components work together in creating a resource for us. Below is the communication flow that happens between the components,



Let's say we are creating a deployment in k8 by running a sample deployment manifest file like below,

```
[root@manja17-I13330 kubenetes-config]# cat basic-deployment.yml
apiVersion: apps/v1beta1
kind: Deployment
metadata:
  name: test-service-deploy
spec:
  replicas: 2
  template:
    metadata:
      labels:
        app: test-service
    spec:
      containers:
      - name: test-ser
        image: docker.io/jagadesh1982/testing-service
        ports:
        - containerPort: 9876
        env:
        - name: VERSION
          value: "0.5"
```

In the above manifest file, we are creating a deployment , replica set and a container by the name test-ser with the image "docker.io/jagadesh1982/testing-service"

1. Before even submitting the request for the above deployment to create , we have controllers , scheduler on control plane and kubelet on worker node are watching for the change updates from Api Server . these components watch for their specific resource changes.
2. Using the kubectl command, we send a HTTP Post request to the Api server. The command is "kubectl create -f <Location of the above manifest file>.
3. The Api server receives the request and perform
   Authentication to authenticate the user
   Authorization to make sure user has enough permissions to create the deployment
   Admission Control check to make sure the request manifest is altered based on certain criteria
   Once all are done successfully. It saves the manifest to the etcd server and sends the response to the kubelet running on the worker node.
4. Now once the data is saved to Etcd, Api server will send a notification to the deployment controller in controllers. The deployment controller watch for events from Api server on deployment resource. This then create a Deployment object and create a Replica set and sends back the details to the Api server.
5. Once the replica set is created, a replication controller is triggered from the controller which in-turn read the replica set and identify the pod details. Api server will in-turn sends a notification to the Replication controller.
6. Once the pod details are observed , pods are created in the Api server. The scheduler then comes into picture.
7. Api server sends a notification to scheduler that pod need to be created. The scheduler will in turn contact Api server for all node details in the cluster from Etcd and based on algorithms it finds suitable nodes to be used for hosting the pods. The details are sent back to the Api server
8. Once the pods details along with node details are obtained, based on the nodes the corresponding kubelet on that node is sent with a notification with the pod details.
9. Once the kubelet receives the pod details, it calls the underlying container runtime to download the image and start the container.

Control Plane or Master Node Configuration

For installing Kubernetes , we will be doing that on 2 machines. The Machine details include
10.131.175.138 - Master Node
172.16.202.96 - Worker Node

Configuration on Master Node

Set the Hostname for the master node as 'k8s-master'

```
hostnamectl set-hostname 'k8s-master'
```

Disable Selinux

```
setenforce 0
sed -i --follow-symlinks 's/SELINUX=enforcing/SELINUX=disabled/g' /etc/sysconfig/selinux
```

Configure the Firewall rules

```
firewall-cmd --permanent --add-port=6443/tcp
firewall-cmd --permanent --add-port=2379-2380/tcp
firewall-cmd --permanent --add-port=10250/tcp
firewall-cmd --permanent --add-port=10251/tcp
firewall-cmd --permanent --add-port=10252/tcp
firewall-cmd --permanent --add-port=10255/tcp
firewall-cmd --reload
```

Set the bridge-nf-call-iptables to 1

```
modprobe br_netfilter
echo '1' > /proc/sys/net/bridge/bridge-nf-call-iptables
```

These control whether or not packets traversing the bridge are sent to iptables for processing

Disable the Swap

```
swapoff -a
```

Swapping results in moving data to and fro from memory. The idea of kubernetes is that all deployments should be pinned with the actual memory/Cpu limits. Swapping a pod details can result in slowness.

Configure the Kubernetes Yum Repo

Create a kubernetes.repo file under /etc/yum.repos.d/ with the below contents,

```
[kubernetes]
name=Kubernetes
baseurl=https://packages.cloud.google.com/yum/repos/kubernetes-el7-x86_64
enabled=1
gpgcheck=1
```

repo_gpgcheck=1
gpgkey=https://packages.cloud.google.com/yum/doc/yum-key.gpg
https://packages.cloud.google.com/yum/doc/rpm-package-key.gpg

Install , Enable & Start Docker and kubeadm

yum install kubeadm docker -y
systemctl restart docker && systemctl enable docker
systemctl restart kubelet && systemctl enable kubelet

Initialize the Kubernetes Master by kubeadm

Run the "kubeadm init" command to initialize the kubernetes master , if all goes well will be seeing an output similar to the below

```
Your Kubernetes master has initialized successfully!

To start using your cluster, you need to run the following as a regular user:

  mkdir -p $HOME/.kube
  sudo cp -i /etc/kubernetes/admin.conf $HOME/.kube/config
  sudo chown $(id -u):$(id -g) $HOME/.kube/config

You should now deploy a pod network to the cluster.
Run "kubectl apply -f [podnetwork].yaml" with one of the options listed at:
  https://kubernetes.io/docs/concepts/cluster-administration/addons/

You can now join any number of machines by running the following on each node
as root:

  kubeadm join 10.131.175.138:6443 --token pe7y1r.zk9u6c07g2nlwm3h
--discovery-token-ca-cert-hash
sha256:48dfb2c9eda08aaed84a70011221804c80adcd700e73d870fd12d041b0054641
```

Save the contents of this output since we will be using in connecting the worker nodes with the master node.

Configure the cluster to use as Root user

Once the master is up and running , we then need to configure our cluster to be used by root user. Execute the commands ,

```
mkdir -p $HOME/.kube
cp -i /etc/kubernetes/admin.conf $HOME/.kube/config
```

```
chown $(id -u):$(id -g) $HOME/.kube/config
```

Once the commands are executed , get the pod list using the Kubectl command as ,

```
[root@k8s-master bin]# kubectl  get pods --all-namespaces
NAMESPACE     NAME                                 READY     STATUS    RESTARTS   AGE
kube-system   etcd-k8s-master                      1/1       Running   0          1m
kube-system   kube-apiserver-k8s-master            1/1       Running   0          37s
kube-system   kube-controller-manager-k8s-master   1/1       Running   0          54s
kube-system   kube-dns-86f4d74b45-xg2wh            0/3       Pending   0          1m
kube-system   kube-proxy-2d6g2                     1/1       Running   0          1m
kube-system   kube-scheduler-k8s-master            1/1       Running   0          1m
```

We will be seeing all pods running in the current cluster. Once Pod "dns" is still in pending.

```
[root@k8s-master bin]# kubectl get nodes
NAME         STATUS     ROLES     AGE       VERSION
k8s-master   NotReady   master    1m        v1.10.1
[root@k8s-master bin]# kubectl get pods
No resources found.
```

If we get the node list even, we see that the master is still in "NotReady" state. We need to get the Master to ready state and also the dns pod to running state. We need to deploy the pod network so that containers in other hosts communicate with each other. POD network is the overlay network between worker nodes.

Note - An overlay network is a telecommunications network that is built on top of another network and is supported by its infrastructure. An overlay network decouples network services from the underlying infrastructure by encapsulating one packet inside of another packet.

The network must be deployed before any application. Kube-dns is a internal helper service will not startup before a network is installed. Several projects provide kubernetes pod networks using CNI (Container network interface) and we will be using one from the weave cloud

Run the below commands to get the Pod Network running and master to ready state,

```
export kubever=$(kubectl version | base64 | tr -d '\n')
kubectl apply -f "https://cloud.weave.works/k8s/net?k8s-version=$kubever"
```

Once we execute the above commands , we see that there are couple of service accounts being created. If all goes well and we run the get pods commands as,

```
[root@k8s-master bin]# kubectl get pods --all-namespaces
NAMESPACE     NAME                                   READY     STATUS    RESTARTS   AGE
kube-system   etcd-k8s-master                        1/1       Running   0          5m
kube-system   kube-apiserver-k8s-master              1/1       Running   0          4m
kube-system   kube-controller-manager-k8s-master     1/1       Running   0          5m
kube-system   kube-dns-86f4d74b45-xg2wh              2/3       Running   0          5m
kube-system   kube-proxy-2d6g2                       1/1       Running   0          5m
kube-system   kube-scheduler-k8s-master              1/1       Running   0          5m
kube-system   weave-net-4swbt                        2/2       Running   0          2m
[root@k8s-master bin]#
```

We will be able to see all the Pods running and also the master in Ready State.

```
[root@manja17-I13330 ~]# kubectl get nodes
NAME                STATUS        ROLES         AGE    VERSION
manja17-i13330      Ready         master1d      v1.10.1
```

Now that all are up and running we are done with our master. We need to configure the worker node.

Check for the K8 version using,

```
[root@manja17-I13330 kubenetes-config]# kubectl version
Client Version: version.Info{Major:"1", Minor:"10", GitVersion:"v1.10.2",
GitCommit:"81753b10df112992bf51bbc2c2f85208aad78335", GitTreeState:"clean",
BuildDate:"2018-04-27T09:22:21Z", GoVersion:"go1.9.3", Compiler:"gc",
Platform:"linux/amd64"}
Server Version: version.Info{Major:"1", Minor:"10", GitVersion:"v1.10.2",
GitCommit:"81753b10df112992bf51bbc2c2f85208aad78335", GitTreeState:"clean",
BuildDate:"2018-04-27T09:10:24Z", GoVersion:"go1.9.3", Compiler:"gc",
Platform:"linux/amd64"}
```

Or

```
[root@manja17-I13330 kubenetes-config]# kubectl version --short
Client Version: v1.10.2
Server Version: v1.10.2
```

Worker Node Configuration

Configure the below steps in the worker node as

Set the Hostname for the master node as 'k8s-master'
```
hostnamectl set-hostname 'dev.foohost.vm'
```

Disable Selinux
```
setenforce 0
sed -i --follow-symlinks 's/SELINUX=enforcing/SELINUX=disabled/g' /etc/sysconfig/selinux
```

Configure the Firewall rules
```
firewall-cmd --permanent --add-port=10250/tcp
firewall-cmd --permanent --add-port=10255/tcp
firewall-cmd --permanent --add-port=30000-32767/tcp
firewall-cmd --permanent --add-port=6783/tcp
firewall-cmd  --reload
```

Set the bridge-nf-call-iptables to 1
```
modprobe br_netfilter
echo '1' > /proc/sys/net/bridge/bridge-nf-call-iptables
```

Disable the Swap

swapoff -a

Swapping results in moving data to and fro from memory. The idea of kubernetes is that all deployments should be pinned with the actual memory/Cpu limits. Swapping a pod details can result in slowness.

Configure the Kubernetes Yum Repo

Create a kubernetes.repo file under /etc/yum.repos.d/ with the below contents,
```
[kubernetes]
name=Kubernetes
baseurl=https://packages.cloud.google.com/yum/repos/kubernetes-el7-x86_64
enabled=1
gpgcheck=1
repo_gpgcheck=1
gpgkey=https://packages.cloud.google.com/yum/doc/yum-key.gpg
       https://packages.cloud.google.com/yum/doc/rpm-package-key.gpg
```

Install , Enable & Start Docker and kubeadm
```
yum install kubeadm docker -y
systemctl restart docker && systemctl enable docker
systemctl  restart kubelet && systemctl enable kubelet
```

Join the worker Node with master

Join the worker node to the master node by running the below command that we copied from the master node,

kubeadm join 10.131.175.138:6443 --token pe7y1r.zk9u6c07g2nlwm3h --discovery-token-ca-cert-hash sha256:48dfb2c9eda08aaed84a70011221804c80adcd700e73d870fd12d041b0054641

This command is available in the output of the “kubeadm init” command in the master node.

```
[root@dev vagrant]# kubeadm join 10.131.175.138:6443 --token oy8f54.2yt2f006mzdw3tfb --discovery-token-ca-cert-hash sha
256:768986b4a64c527ed0fb0e527e3adaa445bb806a661220b6d00a89b632b81607
[preflight] Running pre-flight checks.
        [WARNING Service-Kubelet]: kubelet service is not enabled, please run 'systemctl enable kubelet.service'
        [WARNING FileExisting-crictl]: crictl not found in system path
Suggestion: go get github.com/kubernetes-incubator/cri-tools/cmd/crictl
[discovery] Trying to connect to API Server "10.131.175.138:6443"
[discovery] Created cluster-info discovery client, requesting info from "https://10.131.175.138:6443"
[discovery] Requesting info from "https://10.131.175.138:6443" again to validate TLS against the pinned public key
[discovery] Cluster info signature and contents are valid and TLS certificate validates against pinned roots, will use
API Server "10.131.175.138:6443"
[discovery] Successfully established connection with API Server "10.131.175.138:6443"

This node has joined the cluster:
* Certificate signing request was sent to master and a response
  was received.
* The Kubelet was informed of the new secure connection details.

Run 'kubectl get nodes' on the master to see this node join the cluster.
[root@dev vagrant]#
```

We can see that the node has joined the master successfully. Check the node status using the “kubectl” command as,

```
[root@manja17-I13330 ~]# kubectl get nodes
NAME              STATUS    ROLES     AGE       VERSION
dev.foohost.vm    Ready     <none>    1d        v1.10.1
manja17-i13330    Ready     master    1d        v1.10.1
```

As we can see the master and node are connected.

Kubernetes Resources or Objects

K8 objects are persistent entities in the kubernetes system. K8 uses these entities to represent the state of the cluster. We can use these to describe
Creation of pods & other resources
Resources available to applications
Policies on how pods should run etc

To Work with kubernetes objects like to create , modify or delete we will need to access the Api. For this k8 provides us with a command line utility called "kubectl".

Kubectl make the necessary calls to the Api server for all creating, updating and modifying of the components.

Manifest file

Resources in kubernetes are defined in a manifest which is a simple text file. this manifest represents the kubernetes API object. The manifest is defined declarative format.

Declarative configuration means that you write down the desired state of the world in a configuration and then submit that configuration to a service that takes actions to ensure the desired state becomes the actual state.

Kubernetes API server accepts and processes pod manifest before storing them in persistent storage (etcd) . The scheduler also use k8 api to find pods that haven't been scheduled to a node. The scheduler then places the pods on the node based on the resource availability.

The manifest file will be either in yml or Json format. A format for json would like this,

```
[root@manja17-I13330 kubenetes-config]# cat nginx.json
{
  "apiVersion": "v1",
  "kind": "Pod",
  "metadata": {
      "name": "my-nginx",
      "labels": {
            "env": "dev"
      }
  },
  "spec": {
    "containers": [
     {
```

```
      "name": "my-nginx",
      "image": "nginx",
      "ports": [
        {
          "containerPort": 80
        }
      ]
    }
   ]
  }
}
```

A simple format for yml looks like,

```
[root@manja17-I13330 kubenetes-config]# cat basic-pod-in-namespace.yml
apiVersion: v1
kind: Pod
metadata:
  name: sample-testing
spec:
  containers:
  - name: test-ser
    image: docker.io/jagadesh1982/testing-service
    ports:
    - containerPort: 9876
```

Name Spaces

Namespace can be taught like a project where multiple users or team of users can work. A team can create a namespace say "Project1" to deploy their work in containers in the namespace. The namespace can be isolated from other namespaces so that people can work only in their name spaces.

Namespaces are a logical partitioning capability that enables one kubernetes cluster to be used by multiple users, teams of users or a single user with multiple applications.

Each user, team of users or application exists in namespace isolated from other users and application and making to think as if they were sole user of the clusters.

Namespaces provide for a scope of Kubernetes objects. You can think of it as a workspace you're sharing with other users. Many objects such as pods and services are namespaced, while some (like nodes) are not.

To create a namespace , Create a manifest file "development-ns.yml" with the below contents as,

```
kind: "Namespace"
apiVersion: "v1"
metadata:
 name: "development"
 labels:
   name: "development"
```

Once file is available , create a development namespace by passing the above file as an argument to the kubectl command,

```
[root@manja17-I13330 kubetesting]# kubectl create -f development-ns.yml
namespace "development" created
```

Now check the namespace using

```
[root@manja17-I13330 kubetesting]# kubectl get namespaces
NAME              STATUS    AGE
default           Active    1d
development       Active    12s
kube-public       Active    1d
kube-system       Active    1d
```

If we need to create a pod in a namespace we can use the command,

```
[root@manja17-I13330]# kubectl create --namespace=sample-testing -f
basic-pod-in-namespace.yml
pod "sample-testing" created
```

So all other resources like pod, services etc can be created in namespace. To see what in namespace we have to pass the “--namespace=<name>” argument with the command.

Pods

A Pod is a 1 or group of containers having same application running on any worker node. A Pod is a collection of containers. The Pod name goes with the whale theme of docker containers, since group of whales are called pod.

A pod can also be thought as a smallest deployable artifact in a cluster. A Pod represents a collection of application containers and volumes running in the same execution environment. This means all containers defined inside the pod definition will land on the same execution environment.

Why Pod and why Not Containers?

Many times we may think of why pod and why not containers. A Pod is a smallest unit of workload that can be deployed. A Pod can have one container running which will do a specific job but there would be cases where we require pod to have multiple containers.

The taught of pod is very important. A Container should run 1 process as a standard (though it can run multiple process) and what if need to manage the container. What if we need to have some additional information like restart policy of the container, what to do if the container stops, how to check if the container is up and running the liveness probe or perform certain action based on certain things.

For this we have to run the application process and also another process to manage the container. This will cause extra load on the container and application running inside the container. Hence people came up with a Pod taught where multiple containers that are tightly coupled will be running inside a pod and the pod will be managed as a single entity.

One another advantage of using pod is usage of namespaces. Name spaces are one of the major component for the containers and multiple containers running in a pod can always share certain things including network, IPC etc. They can also share the volumes. Because of these properties container can efficiently communicate between themselves in a pod ensuring data locality. This is one of the reason why pod was given the way to run more than one containers.

How is Pod created?
Kubectl command will use the yml file and connect to the kube-apiserver running on 6443. This in turn will connect to the kube-controller-manager which will further connect to the kube-scheduler. this scheduler program connects to worker nodes using kubelet agent and then kubelet agent connects to docker daemon on the worker node and launch the container based on the pod definition in the YML file

The applications running in the POD all use the same network namespace, IP address and Port Space. They communicate with each other using the localhost.

Each POD will have an IP address and has full communication with other physical computers across the network.

A simple POD is created using yml file. create a file pod.yml file and add the below contents as,

```
apiVersion: v1
kind: Pod
metadata:
 name: kube-pod-testing
 labels:
  app: pingpong-java
  region: IN
  rack: r1
  version: "1.1"
spec:
 containers:
  - name: web-server-test-container
    image: docker.io/rajpr01/myapp
    ports:
    - containerPort: 3000
```

In the above file kind, Name, image and containerPort are very important.
Kind - Type here it is Pod.
Name - name of the container
Image - Image of the container
containerPort - port that is being exposed to the outside from container.

Run the "kubectl create -f pod.yml" which will create a Pod.

Get the Pod details using the ,

```
[root@manja17-I13330 kubetesting]# kubectl get pods
NAME                READY     STATUS    RESTARTS   AGE
kube-pod-testing    1/1       Running   0          1d
```

We can check resources for our running pod by,

```
[root@manja17-I13330 kubetesting]# kubectl describe  pod kube-pod-testing
Name:           kube-pod-testing
Namespace:      default
Node:           dev.foohost.vm/10.0.2.15
Start Time:     Fri, 20 Apr 2018 08:18:42 -0400
Labels:         app=pingpong-java
                rack=r1
                region=IN
                version=1.1
Annotations:    <none>
Status:         Running
IP:             10.44.0.1
Containers:
 web-server-test-container:
   Container ID:
docker://7934cb451aa73d727b34181b9ae78454c7304ad78d12eb7fac7c667e27
                    bdb618
   Image:           docker.io/rajpr01/myapp
   Image ID:        docker-pullable://docker.io/rajpr01/myapp@sha256:287844a655b8c205b
                    9099f8c258b5d6370f921a5953f652eeca64524576e45ea
   Port:            3000/TCP
   Host Port:       0/TCP
   State:           Running
     Started:       Fri, 20 Apr 2018 08:19:38 -0400
   Ready:           True
   Restart Count:   0
   Environment:     <none>
   Mounts:
     /var/run/secrets/kubernetes.io/serviceaccount from default-token-r4w74 (ro)
Conditions:
 Type          Status
 Initialized   True
 Ready         True
 PodScheduled  True
Volumes:
```

```
 default-token-r4w74:
   Type:        Secret (a volume populated by a Secret)
   SecretName:  default-token-r4w74
   Optional:    false
QoS Class:       BestEffort
Node-Selectors:  <none>
Tolerations:     node.kubernetes.io/not-ready:NoExecute for 300s
                 node.kubernetes.io/unreachable:NoExecute for 300s
Events:          <none>
```

Connecting to the Pod - Kubernetes allows us to login to the container using exec command similar to the docker exec. The container names are generally assigned in the manifest file or we can get the details when we describe the pod. we can connect to the pod using,

```
[root@manja17-I13330 kubenetes-config]# kubectl exec testing-service -c test-ser -it -- bash
root@testing-service:/usr/src/app# ls
Dockerfile  requirements.txt  testing_service.py
root@testing-service:/usr/src/app# exit
```

the syntax will be "kubectl exec <pod Name> -c <Container Name> -it -- bash".

Run a Command -We can also run the commands without logging into the container using,

```
[root@manja17-I13330 kubenetes-config]# kubectl exec testing-service -c test-ser free
              total        used        free      shared  buffers     cached
Mem:       4046940     3758468      288472       61204        6368     2668700
-/+ buffers/cache:     1083400     2963540
Swap:            0           0           0
```

Restart a Pod

Restarting a pod can be different in here. Lets see how we can restart the pod

```
[root@manja17-I13330 kubenetes-config]# kubectl get  pods
NAME        READY     STATUS    RESTARTS   AGE
myapp-pod   1/1       Running   0          21m
```

```
[root@manja17-I13330 kubenetes-config]# kubectl get pod myapp-pod -n default -o yaml |
kubectl replace --force -f -
pod "myapp-pod" deleted
pod "myapp-pod" replaced
```

```
[root@manja17-I13330 kubenetes-config]# kubectl get pods
NAME          READY     STATUS    RESTARTS   AGE
myapp-pod   1/1         Running     0              1m
```

Labels

Labels are a mechanism that we use to organize objects in kubernetes. A Kubernetes object can be anything from containers ,pods to services. Label is a key-value pair that we can attach to a resource for identification.

Label contain identifying information and are used by the selector queries or within selector section in object definitions. Lets create a pod with basic labels and see how it works,

```
[root@manja17-l13330 kube-testing]# cat basic-label-pod.yml
apiVersion: v1
kind: Pod
metadata:
 name: testing-service
 labels:
   env: dev
spec:
 containers:
   - name: test-ser
     image: docker.io/jagadesh1982/testing-service
     ports:
     - containerPort: 9876
```

From the above config we are actually creating a pod with label "env=dev". To see the label available for a pod we can use the command,

```
[root@manja17-l13330 kube-testing]# kubectl get pods --show-labels
NAME                READY     STATUS    RESTARTS   AGE       LABELS
sample-test-c9lcw   1/1       Running   0          1d        app=sample-test,version=2
testing-service     1/1       Running   0          50s       env=dev
```

If we want to assign a label while a pod is running, we can use

```
[root@manja17-l13330 kube-testing]# kubectl label pod testing-service owner=kube
pod "testing-service" labeled

[root@manja17-l13330 kube-testing]# kubectl get pods --show-labels
NAME                READY     STATUS    RESTARTS   AGE       LABELS
sample-test-c9lcw   1/1       Running   0          1d        app=sample-test,version=2
testing-service     1/1       Running   0          1m        env=dev,owner=kube
```

Selector in kubernetes allow to use labels for filtering , for example to list pods that have an owner is equal to kube

```
[root@manja17-I13330 kube-testing]# kubectl get pods --selector owner=kube
NAME              READY     STATUS    RESTARTS   AGE
testing-service   1/1       Running   0          2m
```

The `--selector` option can be abbreviated to `-l`, so to select pods that are labelled with `env=dev`,

```
[root@manja17-I13330 kube-testing]# kubectl get pods -l env=dev
NAME              READY     STATUS    RESTARTS   AGE
testing-service   1/1       Running   0          2m
```

To get all pods that are labelled with dev or test

```
[root@manja17-I13330 kube-testing]# kubectl get pods -l 'env in (dev,test)'
NAME              READY     STATUS    RESTARTS   AGE
testing-service   1/1       Running   0          3m
```

Annotations

Annotations tend to be used for added automation of kubernetes. Some use cases where you want to add an annotation are

Build, release, or image information like timestamps, release IDs, git branch, PR numbers, image hashes, and registry address.

Client library or tool information that can be used for debugging purposes: for example, name, version, and build information.

```
[root@manja17-I13330 kubenetes-config]# kubectl annotate pods testing-service-s8xzt description="simple annotation"
pod "testing-service-s8xzt" annotated

[root@manja17-I13330 kubenetes-config]# kubectl describe pod testing-service-s8xzt
Name:           testing-service-s8xzt
Namespace:      default
Node:           manja17-i14021/10.131.36.181
Start Time:     Fri, 27 Jul 2018 05:19:06 -0400
Labels:         app=testingService-service
Annotations:    description=simple annotation
Status:         Running
IP:             10.38.0.6
Controlled By:  ReplicationController/testing-service
******************************
```

Nodes

A node is basically a worker machine where kubernetes creates pods. The node can be a VM or physical machine. Every node in the cluster will have some services running which include docker, kubelet and kube-proxy

The node is not generally created through kubernetes but by cloud providers or Vm based softwares. Once created and attached to kubernetes cluster , a node object is created which will be used to check if the node is valid or not. If the node is valid and all necessary worker node services are running, then the node is eligible for pod creation. If the node is not valid ,it will wait until the node become valid to create pod.

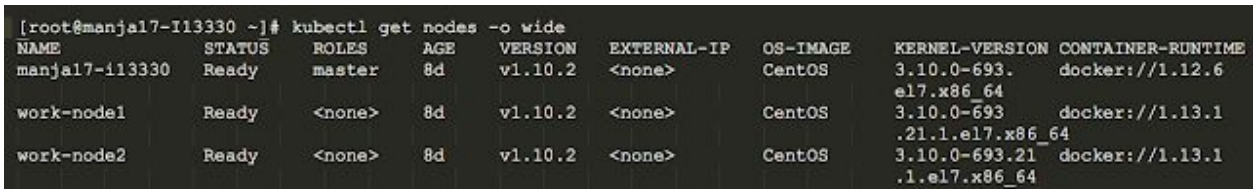
Who will handle the nodes?

Node controller is the master node component which take care of the nodes. It has some of the responsibilities including node health check also making sure all nodes in the node controller list are up to date.

To get the nodes available in the cluster, use

```
[root@manja17-I13330 ~]# kubectl get nodes
NAME            STATUS   ROLES    AGE     VERSION
manja17-i13330  Ready    master   8d      v1.10.2
work-node1      Ready    <none>   8d      v1.10.2
work-node2      Ready    <none>   8d      v1.10.2
```

To get more details about the node use,



To get more insight into the node , we can use the describe command as,

```
[root@manja17-I13330 ~]# kubectl describe node work-node1
Name:               work-node1
Roles:              <none>
Labels:             app=backend
                    beta.kubernetes.io/arch=amd64
                    beta.kubernetes.io/os=linux
                    kubernetes.io/hostname=work-node1
                    label=node1
```

Annotations: node.alpha.kubernetes.io/ttl=0
volumes.kubernetes.io/controller-managed-attach-detach=true
CreationTimestamp: Tue, 01 May 2018 09:19:56 -0400
Taints: <none>
Unschedulable: false
Conditions:
Type Status LastHeartbeatTime LastTransitionTime Reason Message
---- ------ ----------------- ------------------ ------ -------
OutOfDisk False Thu, 10 May 2018 00:40:10 -0400 Tue, 01 May 2018 09:19:55 -0400
KubeletHasSufficientDisk kubelet has sufficient disk space available

MemoryPressure False Thu, 10 May 2018 00:40:10 -0400 Tue, 01 May 2018 09:19:55
-0400 KubeletHasSufficientMemory kubelet has sufficient memory available

DiskPressure False Thu, 10 May 2018 00:40:10 -0400 Tue, 01 May 2018 09:19:55 -0400
KubeletHasNoDiskPressure kubelet has no disk pressure

PIDPressure False Thu, 10 May 2018 00:40:10 -0400 Tue, 01 May 2018 09:19:55 -0400
KubeletHasSufficientPID kubelet has sufficient PID available

Ready True Thu, 10 May 2018 00:40:10 -0400 Tue, 01 May 2018 09:20:45 -0400
KubeletReady kubelet is posting ready status
Addresses:
InternalIP: 10.0.2.15
Hostname: work-node1
Capacity:
cpu: 1
ephemeral-storage: 39269648Ki
hugepages-2Mi: 0
memory: 1883484Ki
pods: 110
Allocatable:
cpu: 1
ephemeral-storage: 36190907537
hugepages-2Mi: 0
memory: 1781084Ki
pods: 110
System Info:
Machine ID: d43fcbfca85f47288d8b553380d86c8e

System UUID: D43FCBFC-A85F-4728-8D8B-553380D86C8E
Boot ID: 324be3e3-a221-46d9-b724-50f2daa061c2
Kernel Version: 3.10.0-693.21.1.el7.x86_64
OS Image: CentOS Linux 7 (Core)
Operating System: linux
Architecture: amd64
Container Runtime Version: docker://1.13.1
Kubelet Version: v1.10.2
Kube-Proxy Version: v1.10.2
ExternalID: work-node1
Non-terminated Pods: (3 in total)

Namespace	Name	CPU Requests	CPU Limits	Memory Requests	Memory Limits
default	myapp-pod	0 (0%)	0 (0%)	0 (0%)	0 (0%)
kube-system	kube-proxy-9qpxq	0 (0%)	0 (0%)	0 (0%)	0 (0%)
kube-system	weave-net-frhh9	20m (2%)	0 (0%)	0 (0%)	0 (0%)

Allocated resources:
(Total limits may be over 100 percent, i.e., overcommitted.)

CPU Requests	CPU Limits	Memory Requests	Memory Limits
20m (2%)	0 (0%)	0 (0%)	0 (0%)

Services

We Know that container running in pod will share same network space. Every pod will have an IP address. Once the Pod is killed or destroyed a new pod is created by replication controller which will get new IP address. How can an user still access the application with changing IP address (Container getting killed and recreated)?

This is where service comes to help. Service provides a single name for a set of pods.They basically act as a load balancer to a set of same application running in different pods. Each service will be assigned with a virtual IP address which remains constant until the service life time.

The service created is attached to the application/pods. User will be accessing the service address rather the pod address/IP address. The service will keep track of the pds being created and destroyed and the load will be shared with them.

Create a Service yml file for the pod that we created earlier using,
[root@manja17-I13330 kubenetes-config]# cat testing-service-pod.yml

```
apiVersion: v1
kind: ReplicationController
metadata:
 name: testing-service
spec:
 replicas: 1
 selector:
   app: testingService-service
 template:
   metadata:
     name: testingService
     labels:
       app: testingService-service
   spec:
     containers:
     - name: test-ser
       image: docker.io/jagadesh1982/testing-service
       ports:
       - containerPort: 9876
```

Create the service using the
[root@manja17-I13330 kubenetes-config]# cat testingService-service.yml

```
apiVersion: v1
kind: Service
metadata:
 name: simpleservice
spec:
 ports:
   - port: 80
     targetPort: 9876
 selector:
   app: testingService-service
```

In the above service we are actually creating a service or endpoint to access for the pod with label set as "app=testingService-service". Though the pod endpoint to access the application is 9876, we created a service with endpoint 80. So when we access the service on port 80 ,it will routes to the application on port 9876.

Check for the pods are created,

```
[root@manja17-I13330 kubenetes-config]# kubectl get pods
NAME                    READY    STATUS    RESTARTS   AGE
testing-service-v2kk4   1/1      Running   0           51s
```

Now lets see if the service is created are not,

```
[root@manja17-I13330 kubenetes-config]# kubectl get svc
NAME            TYPE        CLUSTER-IP     EXTERNAL-IP   PORT(S)   AGE
kubernetes      ClusterIP   10.96.0.1      <none>        443/TCP   13d
simpleservice   ClusterIP   10.97.14.179   <none>        80/TCP    4s
```

Now we can access the application using,

```
[root@manja17-I13330 kubenetes-config]# curl 10.97.14.179:80/info
{"host": "10.97.14.179", "version": "0.5.0", "from": "10.32.0.1"}
```

Because every node in a Kubernetes cluster runs a kube-proxy, the kube-proxy watches the Kubernetes API server for the addition and removal of services. For each new service, kube-proxy opens a (randomly chosen) port on the local node. Any connections made to that port are proxied to one of the corresponding backend pods.

Replication Controller

Replication controller manage the life cycle of a POD. A Pod by itself will not have any lifecycle. This replication controller will ensure that specified number of pods are running at any given time. They do this by creating or deleting pods as required.

Lets say if i want to run 3 application containers, i will define the replica for the container to be 3. K8 will make sure that at any point of time 3 containers are up and running. If any one of them by any chance are terminated, K8 will get another one up and running.

Create a manifest file with replication controller definition as,

```
[root@manja17-I13330 kubenetes-config]# cat basic-replicaController.yml
apiVersion: v1
kind: ReplicationController
metadata:
 name: testing-service
spec:
 replicas: 2
 template:
   metadata:
     labels:
       app: test-service
       version: "2"
   spec:
     containers:
       - name: test-ser
         image: "docker.io/jagadesh1982/testing-service"
         ports:
          - containerPort: 9876
```

In the above config file, we are telling to create a replication controller which should have 2 replicas of pods always running.

Create the replication controller using,

```
[root@manja17-I13330 kubenetes-config]# kubectl create -f basic-replicaController.yml
replicationcontroller "testing-service" created
```

Once i create the rc and see the pod details, we can see that 2 pods are being created.

```
[root@manja17-I13330 kubenetes-config]# kubectl get pods
NAME                   READY     STATUS              RESTARTS   AGE
testing-service-bmtsc  0/1       ContainerCreating   0          4s
```

```
testing-service-xkh6k   0/1       ContainerCreating   0             4s
```

To check the replication controller details we can use the below command as,

```
[root@manja17-I13330 kubenetes-config]# kubectl get rc
NAME              DESIRED   CURRENT   READY     AGE
testing-service   1         1         1         6s
```

If we want to scale the replica we can use,

```
[root@manja17-I13330 kubetesting]# kubectl scale --replicas=3 rc testing-service
replicationcontroller "my-myapp" scaled
```

This makes sure that at all time 3 replicas of the pod will be running. Check the replication controller using,

```
[root@manja17-I13330 kubenetes-config]# kubectl get rc
NAME              DESIRED   CURRENT   READY     AGE
testing-service   3         3         3         3m
```

Now we can see that another pod will be created automatically. Now if we delete a running pod ,

```
[root@manja17-I13330 kubenetes-config]# kubectl get pods
kNAME                   READY     STATUS    RESTARTS   AGE
testing-service-5tqxn   1/1       Running   0          47s
testing-service-bmtsc   1/1       Running   0          3m
testing-service-xkh6k   1/1       Running   0          3m
```

```
[root@manja17-I13330 kubenetes-config]# kubectl delete pod testing-service-xkh6k
pod "testing-service-xkh6k" deleted
```

And check for the pods, we can see that the pod we deleted is being terminating and another new pod is being created. This creation is now automatically done by the replication Controller

```
[root@manja17-I13330 kubenetes-config]# kubectl get pods
NAME                    READY     STATUS              RESTARTS   AGE
Testing-service-5tqxn   1/1       Running             0          57s
testing-service-bmtsc   1/1       Running             0          4m
testing-service-vbmkh   0/1       ContainerCreating   0          2s
testing-service-xkh6k   1/1       Terminating         0          4m
```

ReplicaSets are the next generation Replication controller which will be used and are based on Set based label selector.

Replica Set

A Replica Set is a term for API objects in K8 that refer to pod replicas. The basic idea of Replica Set is to control a set of Pod behavior. The Replica Set makes sure that a specified number of pods will always be running. If somehow one of the pod is crashed, another one will start automatically.

Replication Controller vs Replica Set
Both Replica Set and Replication Controller does the same thing. From version 1.2, Replica set does the job as Replication Controller. K8 even now allows to create replication controller now but it is advised to use Replica Set since they are up to date and have finer granularity of configuration

```
[root@manja17-I13330 kubenetes-config]# cat basic-replicaSet.yml
apiVersion: extensions/v1beta1
kind: ReplicaSet
metadata:
  name: testing-service
  labels:
    version: 0.0.1
spec:
  replicas: 3
  template:
    metadata:
      labels:
        role: frontend
        env: dev
    spec:
      containers:
        - name: test-ser
          image: "docker.io/jagadesh1982/testing-service"
          ports:
           - containerPort: 9876
```

The important element in the above configuration is the “replicas: 3”. This tells that at least 3 replicas of the container should be running all the time

Create the replica Set using,

```
[root@manja17-I13330 kubenetes-config]# kubectl create -f basic-replicaSet.yml
replicaset.extensions "testing-service" created
```

Now if we see pod list, we see that there are 3 pods running

```
[root@manja17-I13330 kubenetes-config]# kubectl get pods
NAME                          READY     STATUS    RESTARTS   AGE
pod/testing-service-4gbkf     1/1       Running   0          10s
pod/testing-service-649jc     1/1       Running   0          10s
pod/testing-service-pcxwx     1/1       Running   0          10s
```

Check the replica set using,

```
[root@manja17-I13330 kubenetes-config]# kubectl get rs
NAME                              DESIRED   CURRENT   READY     AGE
replicaset.apps/testing-service   3         3         3         10s
```

We can edit the replica set using,

```
[root@manja17-I13330 kubenetes-config]# kubectl edit rs testing-service
replicaset.extensions "testing-service" edited
```

I changed the replicas from 3 to 4. Once you edit the replica set, a new pod will be created automatically,

```
[root@manja17-I13330 kubenetes-config]# kubectl get pods
NAME                    READY     STATUS              RESTARTS   AGE
testing-service-4gbkf   1/1       Running             0          9m
testing-service-649jc   1/1       Running             0          9m
testing-service-lh4l5   0/1       ContainerCreating   0          3s
testing-service-pcxwx   1/1       Running             0          9m
```

Daemon Sets

Kubernetes has a concept of ReplicaSets which will allow to scale the pods based on the requirements. The pods can be created on any nodes without special conditions. but what if we need to run a pod on every node that has a special condition Or what if you want to run a pod on every node that can do some special action like log collectors and monitoring agents

Daemon sets in kubernetes does the same job. An example would be a log collector pod which needs to run on every worker-node that we have and we add. So we attach a label to the node like "app=frontend-node" and configure the pod as a daemon set saying to run this on every node that have a label "app=frontend-node". When we add a new node to the cluster all we have to do is to attach a label to that as "app=frontend-node". DaemonSets share similar functionality with ReplicaSets; both create Pods that are expected to be long-running services and ensure that the desired state and the observed state of the cluster match. So what is the difference between Replicaset and a Daemon set?

Rs can be used when we have an application that is completely decoupled with the node and can run multiple copies on any given node without special conditions. on the other side Daemonset can be used when we want our application pod to run on subset of nodes in our cluster based on the conditions like app=frontend-node. Create a DaemonSet as below,
[root@manja17-I13330 kubenetes-config]# cat basic-daemonset.yml

```
apiVersion: extensions/v1beta1
kind: DaemonSet
metadata:
 name: frontend
spec:
 template:
   metadata:
     labels:
       app: frontend-webserver
   spec:
     nodeSelector:
       app: frontend-node
     containers:
       - name: webserver
         image: nginx
         ports:
         - containerPort: 80
```

In the above Yaml file , we are creating a daemon set with the name “frontend”. The yaml itself has 2 main things node selector and container spec. we are very much aware of the container spec , the node selector tells kubernetes which nodes are part of the condition and should run containers. In other words we are using the label “app=frontend-node” to define on certain nodes in the cluster. Once the nodes are labeled with the frontend-node label, the pod defined with container web server will automatically run.

Label a node first with the “app=frontend-node”
kubectl label node work-node1 app=frontend-node
kubectl label node work-node2 app=frontend-node

In the above command iam defining my machine work-node1 and work-node2 with the label “app=frontend-node” . once you are done this check the labels using “kubectl get nodes --show-labels”.

Create the Daemonset using the above yaml file
[root@manja17-I13330 kubenetes-config]# kubectl create -f basic-daemonset.yml
daemonset.extensions "frontend" created

Check the daemonsets
[root@manja17-I13330 kubenetes-config]# kubectl get ds
NAME DESIRED CURRENT READY UP-TO-DATE AVAILABLE NODE SELECTOR AGE
frontend 2 2 0 2 0 app=frontend-node 4s

Now when we try to get the pods ,we see 2 pods with the frontend are automatically being created ,
[root@manja17-I13330 kubenetes-config]# kubectl get pods
NAME READY STATUS RESTARTS AGE
frontend-7sc22 0/1 ContainerCreating 0 7s
frontend-v6xvq 0/1 ContainerCreating 0 7s

Since we have 2 nodes with the condition “app=frontend-node”, we see 2 pods creating. Lets change the label for one of the node and see what happens
[root@manja17-I13330 kubenetes-config]# kubectl label node work-node1 --overwrite app=backend
node "work-node1" labeled

[root@manja17-I13330 kubenetes-config]# kubectl get pods
NAME READY STATUS RESTARTS AGE
frontend-7sc22 1/1 Running 0 18m
We can see that one of the pods was automatically deleted.

Deployments

We can always create a Replication controller or replica set, so that it will take care of how many pods it should keep running. What if we need to do a rolling update to the pod running?. What if we have new changes and want to push them to pod?

Deployment is the one which provide these. A Deployment is a supervisor for pods and replica sets giving you a fine grained control over how and when a new pod version is rolled out as well as rolled back to the previous state. The main purpose is to keep a set of identical pods running and update them in a controlled way, performing a rolling update by default.

Create a basic deployment as,

```
[root@manja17-I13330 kubenetes-config]# cat basic-deployment.yml
apiVersion: apps/v1beta1
kind: Deployment
metadata:
 name: test-service-deploy
spec:
 replicas: 2
 template:
   metadata:
     labels:
       app: test-service
   spec:
     containers:
     - name: test-ser
       image: docker.io/jagadesh1982/testing-service
       ports:
       - containerPort: 9876
       env:
       - name: VERSION
         value: "0.5"
```

We are actually creating a basic deployment for the testing-service. We also passed and environment value called “Version” with a value “0.5”. Most of the other elements are same.

Check if the deployments are available,

```
[root@manja17-I13330 kube-testing]# kubectl get deployments
NAME                 DESIRED   CURRENT   UP-TO-DATE   AVAILABLE   AGE
test-service-deploy   2         2         2            0           6s
```

Now if we see that we have 2 pods created automatically.

```
[root@manja17-I13330 kube-testing]# kubectl get pods
NAME                                  READY     STATUS    RESTARTS   AGE
test-service-deploy-85c8787b6f-6msvv   1/1       Running   0          13s
test-service-deploy-85c8787b6f-nc9kz   1/1       Running   0          13s
```

Now lets take a POD ,get the IP and see if we can access the application

```
[root@manja17-I13330 kube-testing]# kubectl describe pod
test-service-deploy-85c8787b6f-6msvv | grep IP
IP:           10.47.0.1

[root@manja17-I13330 kube-testing]# curl 10.47.0.1:9876/info
{"host": "10.47.0.1:9876", "version": "0.5", "from": "10.32.0.1"}
```

If we see that we have accessed the application and we can see that the version is 0.5 in here. Now if we want to make any modifications ,we can edit the deploy file using,

```
[root@manja17-I13330 kube-testing]# kubectl edit deploy/test-service-deploy
deployment.extensions "test-service-deploy" edited
```

I made the replicas from 2 to 3 and saved the changes. I also changed the env value from "0.5" to "2". This way i am actually pushing new changes to my container code.

```
[root@manja17-I13330 kube-testing]# kubectl get deploy
NAME                 DESIRED   CURRENT   UP-TO-DATE   AVAILABLE   AGE
test-service-deploy   2         3         1            2           5m
```

Now if we see the pods we can see,

```
[root@manja17-I13330 kube-testing]# kubectl get pods
NAME                                  READY     STATUS        RESTARTS   AGE
test-service-deploy-5fdf56d98c-hvsjn   1/1       Terminating   0          2m
test-service-deploy-5fdf56d98c-sglr8   1/1       Terminating   0          2m
test-service-deploy-98bf447c8-c7ct6    1/1       Running       0          14s
test-service-deploy-98bf447c8-lh7s6    1/1       Running       0          8s
```

Get the IP address for one of the POD and access the application as,

```
[root@manja17-I13330 kube-testing]# kubectl describe pod test-service-deploy-98bf447c8-c7ct6 | grep IP
IP:           10.44.0.1

[root@manja17-I13330 kube-testing]# curl http://10.44.0.1:9876/info
{"host": "10.44.0.1:9876", "version": "2", "from": "10.32.0.1"}
```

We can see that the application is now pointing to the version 2 as we passed in our environment variables. Beside providing a single point for your whole deployment , it also provides some more features like,

```
[root@manja17-I13330 kube-testing]# kubectl rollout history deploy/test-service-deploy
deployments "test-service-deploy"
REVISION  CHANGE-CAUSE
1         <none>
2         <none>
3         <none>
```

We can see the rollout version of the deployment. We can also check the progress of the deployment using,

```
[root@manja17-I13330 kube-testing]# kubectl rollout status deploy/test-service-deploy
deployment "test-service-deploy" successfully rolled out
```

We can also rollback to the old version using,

```
[root@manja17-I13330 kube-testing]# kubectl rollout undo deploy/test-service-deploy --to-revision=1
deployment.apps "test-service-deploy"
```

And once the rollback is done, we can access the application and see the application is pointing to the old version as below,

```
[root@manja17-I13330 kube-testing]# curl 10.47.0.3:9876/info
{"host": "10.47.0.3:9876", "version": "0.5", "from": "10.32.0.1"}
```

Health Checks

Health Checks are exactly what they sound like , a way of checking the health of the resource. In the case of a pod, a health check is used to determine the health of a running container.

When a health check is specified, it gives us a way to test if the container is working or not,basically to test if the services are up or not. With no health check specified, it can be complex to find if the service running inside a container is active or not.

Kubernetes provides 2 layers of health checking Http , Tcp checks and application checks.
Http or Tcp - In this type of checking K8 can attempt to connect to the particular endpoint that is provided in the check and gives a status of health on the successfully connection.

Application Specific - In this type of health check , application health is identified by using command line scripts.

Http Check : In this we are going to create a http url hit and see if the pod endpoint can be connected or not

Create a basic health check pod as,

```
[root@manja17-I13330 kubenetes-config]# cat health-httpGet-pod.yml
apiVersion: v1
kind: Pod
metadata:
 name: testing-service
spec:
 containers:
   - name: test-ser
     image: docker.io/jagadesh1982/testing-service
     ports:
     - containerPort: 9876
     livenessProbe:
       initialDelaySeconds: 2
       periodSeconds: 5
       httpGet:
         path: /info
         port: 9876
```

In the above yaml file, we have created a health check to let k8 connect to the endpoint "/info". The important thing is the livenessProbe element. Under this we can specify either httpGet,tcpSocket or exec. K8 will check the path and port specified and restart the pod if it does not successfully return. Generally a status code between 200 to 399 are all considered as healthy. The "initialDelaySeconds" lets us to wait until the pod finished initializing so that K8 can probe a health check.

Create the pod and confirm ,

```
[root@manja17-I13330 kubenetes-config]# kubectl get pods
kNAME            READY     STATUS    RESTARTS   AGE
testing-service   1/1         Running   0              27s
```

Now describe the pod using,

```
[root@manja17-I13330 kubenetes-config]# kubectl describe pod testing-service
Name:         testing-service
Namespace:    default
Node:         work-node1/10.0.2.15
Start Time:   Fri, 18 May 2018 03:47:03 -0400
Labels:       <none>
Annotations:  <none>
Status:       Running
IP:           10.44.0.1
Containers:
  Liveness:       http-get http://:9876/info delay=2s timeout=1s period=5s #success=1 #failure=3
 ************
```

Health Check - Exec : In this type of health check we will use k8 application specific or a command to identify the health of a container.

Create a exec health check as,

```
[root@manja17-I13330 kubenetes-config]# cat health-exec-pod.yml
apiVersion: v1
kind: Pod
metadata:
 name: testing-service
spec:
 containers:
   - name: test-ser
```

```
    image: docker.io/jagadesh1982/testing-service
    ports:
    - containerPort: 9876
    livenessProbe:
      initialDelaySeconds: 2
      periodSeconds: 5
      exec:
       command:
        - cat
        - /testing_service.py
```

The period seconds field specifies that the k8 should perform the liveness probe every 5 seconds. initialDelaySeconds let the container complete initializing before the liveness test is done.
The exec probe executes the command "cat /testing_service.py" in the container. If the file is available and command is success full which mean returns 0 then health check is success. If the command returns non-zero , k8 will kill the containers and restart.

Health Check - Tcp Sockets: A health check for a TCP Port check or validation can also be done in the Pod as below,

A Tcp Health check can be configured using,
[root@manja17-I13330 kubenetes-config]# cat health-tcpSocket-pod.yml

```
apiVersion: v1
kind: Pod
metadata:
 name: testing-service
spec:
 containers:
   - name: test-ser
     image: docker.io/jagadesh1982/testing-service
     ports:
     - containerPort: 9876
     livenessProbe:
       initialDelaySeconds: 2
       periodSeconds: 5
       tcpSocket:
         port: 9876
```

This will check the port 9876, it that is up and used.

Resource Management - Allocation & Quotas

A Capacity Planning Is used to determine the resources needed in order to meet the future workflow demands. In Kubernetes resource management is very important. We learned that scheduler is the component that identifies best nodes for creating pods.

The Scheduler has to take into account many factors when it is scheduling pods on nodes. If there are overlapping details or conflicting details the scheduler can find problem when allocating pods in nodes.

Let's say if we nave a daemon set that is running on all nodes which require 50% of memory, then scheduler can find it hard to allocate pods that would require around same memory. Problems of this type can easily mitigated by using namespace resource quotas and correct management of the resource like Cpu, memory and storage

Kubernetes out of box supports resource quotas. The Admission control element in the API server makes sure to enforce resource quotas when one is defined. Kubernetes enforces one resource quota per name space.

The Resource Quotas in Kubernetes are 3 types
Compute
Storage
Object

Compute Resource Quota
This Quota talks about compute resources which are memory and Cpu. In this we can either request for a quota or limit a quota. This can be done by using limits.[cpu,memory] and requests.[cpu,memory]

Limits.cpu & limits.memory – This values talk about the sum of the CPU and Memory limits across all pods that cannot be exceeded.

Requests.cpu & requests.memory – This value talks about the sum of CPU and memory requests by all pods that cannot be exceeded

Storage Resource Quota
This storage resource quota talks about resource that can be restricted per namespace. The resource includes the amount of storage and persistent volume claims.

Object Resource Quota

The Object resource quota talks about the limitations on objects that can be created per namespace. The goal of this resource quota is to make sure that Api server is not wasting time performing things that are useless.

Compute Quota

Lets create a simple compute quota and see how things go. In the below manifest file we are limiting the number of pods to 2. When this quota is attached to a namespace, kubernetes blocks creation of the 3 pods in the namespace.

Let's see how we can allocate resources to the pods,

```
[root@manja17-l13330 kubenetes-config]# cat basic-single-container-pod.yml
apiVersion: v1
kind: Pod
metadata:
 name: testing-service
spec:
 containers:
  - name: test-ser
    image: docker.io/jagadesh1982/testing-service
    ports:
    - containerPort: 9876
    resources:
      limits:
        memory: "64Mi"
        cpu: "500m"
```

Once we create the pod using the kubectl command and describe the pod we can see the limits defined as,

```
Limits:
    cpu:     500m
    memory:  64Mi
Requests:
    cpu:        500m
    memory:     64Mi
```

So we have the limits defined as requested and allocated.

Quota - A resource quota, defined by a ResourceQuota object, provides constraints that limit aggregate resource consumption per namespace. It can limit the quantity of objects that can be created in a namespace by type, as well as the total amount of compute resources that may be consumed by resources in that project.

Create a resource quota config file as

```
apiVersion: v1
kind: ResourceQuota
metadata:
  name: pod-demo
spec:
  hard:
    pods: "2"
```

In our resource quota file we have limited our number of pod creation to 2.

Create the resource quota by assigning to a namespace,

```
[root@manja17-I13330 kubenetes-config]# kubectl create -f basic-resourcequota.yml
--namespace=sample-testing
resourcequota "pod-demo" created
```

We set the resource quote to a name space “sample-testing”. So in this namespace only 2 pods will be allowed to create

Once created , get the resource quota details as,

```
[root@manja17-I13330 kubenetes-config]# kubectl get resourcequota
--namespace=sample-testing
NAME       AGE
pod-demo   55s
```

When we get the pods, we see we already have 2 pods running in this namespace,

```
[root@manja17-I13330 kubenetes-config]# kubectl get pods
--namespace=sample-testing
NAME                                    READY   STATUS    RESTARTS   AGE
test-service-deploy-85c8787b6f-sfprn    1/1     Running   0          57s
test-service-deploy-85c8787b6f-wvnfc    1/1     Running   0          14s
```

Now let's try to create a new pod and see how it goes

```
[root@manja17-I13330 kubenetes-config]# cat basic-single-container-pod.yml
apiVersion: v1
kind: Pod
metadata:
```

```
 name: testing-service
spec:
 containers:
   - name: test-ser
     image: docker.io/jagadesh1982/testing-service
     ports:
     - containerPort: 9876
     resources:
       limits:
         memory: "64Mi"
         cpu: "500m"
```

```
[root@manja17-I13330 kubenetes-config]# kubectl create -f
basic-single-container-pod.yml --namespace=sample-testing
```

```
Error from server (Forbidden): error when creating "basic-single-container-pod.yml":
pods "testing-service" is forbidden: exceeded quota: pod-demo, requested: pods=1,
used: pods=2, limited: pods=2
```

Now when i run the pod creation it gives me an error saying that the resource quota limit for pod is reached and cannot create any more.

Job Management

In Kubernetes , a Job is a controller object that represents a finite task. Generally if we have a replication controller or replica set , we will have the kubernetes engine run a specified number of pods running all the time. They make sure that pods are up and running at any time.

Jobs differ from the other controller objects is that jobs manage the task as it runs to completion rather than managing an ongoing desired state (such as total number of running pods).

The general use case where jobs are used when we need to run some computational jobs or batch oriented tasks. You can use a Job to run independent but related work items in parallel: sending emails, rendering frames, transcoding files, scanning database keys, etc. However, Jobs are not designed for closely-communicating parallel processes such as continuous streams of background processes.

K8 Supports different type of jobs - Simple job, repeatable job , Parallel job and Cron Job

Simple Job - Lets create a simple Job as,

```
[root@manja17-I13330 kubenetes-config]# cat basic-job.yml
apiVersion: batch/v1
kind: Job
metadata:
 name: testing-service
spec:
 template:
   metadata:
     labels:
       name: testing-service
   spec:
     containers:
     - name: counter
       image: centos:7
       command:
         - "bin/bash"
         - "-c"
         - "for i in 9 8 7 6 5 4 3 2 1 ; do echo $i ; done"
     restartPolicy: Never
```

This is a simple job manifest file which when created will create a pod with single container called "counter". the job of the counter container is to execute a command with a bash for loop.

Create the pod and check the pod status as,

```
[root@manja17-I13330 kubenetes-config]# kubectl get pods
NAME                    READY     STATUS      RESTARTS   AGE
testing-service-gfmrz   0/1       Completed   0          31s
```

The pod is created and status is completed which means counter container has completed the job. We can also get the jobs available current using "kubectl get jobs". Lets describe the pod to see what it did,

```
[root@manja17-I13330 kubenetes-config]# kubectl describe job/testing-service
Name:           testing-service
Namespace:      default
Selector:       controller-uid=0e67e17a-5b96-11e8-9494-020055e1ea1d
Labels:         controller-uid=0e67e17a-5b96-11e8-9494-020055e1ea1d
                job-name=testing-service
                name=testing-service
Annotations:    <none>
Parallelism:    1
Completions:    1
Start Time:     Sat, 19 May 2018 14:54:27 -0400
Pods Statuses:  0 Running / 1 Succeeded / 0 Failed
Pod Template:
 Labels:  controller-uid=0e67e17a-5b96-11e8-9494-020055e1ea1d
          job-name=testing-service
          name=testing-service
 Containers:
  counter:
   Image:      centos:7
   Port:       <none>
   Host Port:  <none>
   Command:
     bin/bash
     -c
     for i in 9 8 7 6 5 4 3 2 1 ; do echo $i ; done
   Environment:  <none>
   Mounts:       <none>
```

```
 Volumes:        <none>
Events:
 Type    Reason            Age   From                Message
 ----    ------            ----  ----                -------
 Normal  SuccessfulCreate  2m    job-controller      Created pod: testing-service-gfmrz
```

We can check the logs of the pod to see what exactly happened as,

```
[root@manja17-I13330 kubenetes-config]# kubectl logs testing-service-gfmrz
9
8
7
6
5
4
3
2
1
```

Repeatable Job - There may be cases where you want to run a program repeatedly. It is very use full to run a program at certain times repeatedly to check something or perform some other actions

Lets create a basic repeatable job manifest file,

```
[root@manja17-I13330 kubenetes-config]# cat basic-repeatable-job.yml
apiVersion: batch/v1
kind: Job
metadata:
  name: testing-service
spec:
  completions: 3
  template:
    metadata:
      labels:
        name: testing-service
    spec:
      containers:
      - name: counter
        image: centos:7
        command:
          - "bin/bash"
          - "-c"
          - "for i in 9 8 7 6 5 4 3 2 1 ; do echo $i ; done"
      restartPolicy: Never
```

The important element is the completions: 3 which tells to run the job until completed for 3 times

```
[root@manja17-I13330 kubenetes-config]# kubectl create -f basic-repeatable-job.yml
job.batch "testing-service" created

[root@manja17-I13330 kubenetes-config]# kubectl get pods
NAME                    READY     STATUS              RESTARTS   AGE
testing-service-strtz   0/1       ContainerCreating   0          3s

[root@manja17-I13330 kubenetes-config]# kubectl get pods
NAME                    READY     STATUS              RESTARTS   AGE
testing-service-dpwr2   0/1       ContainerCreating   0          2s
testing-service-strtz   0/1       Completed           0          6s

[root@manja17-I13330 kubenetes-config]# kubectl get pods
NAME                    READY     STATUS      RESTARTS   AGE
testing-service-7vrx5   0/1       Completed   0          4s
testing-service-dpwr2   0/1       Completed   0          6s
testing-service-strtz   0/1       Completed   0          10s
```

If we see the above pods, we see that they are created one after another and as defined ran a bash command. Lets see the job status ,

```
[root@manja17-I13330 kubenetes-config]# kubectl get jobs
NAME              DESIRED   SUCCESSFUL   AGE
testing-service   3         3            1m
```

Now if we describe the job, we can see

```
[root@manja17-I13330 kubenetes-config]# kubectl describe job testing-service
Name:           testing-service
Namespace:      default
Selector:       controller-uid=be5f8b69-8f0c-11e8-a791-020055e1ea1d
Labels:         controller-uid=be5f8b69-8f0c-11e8-a791-020055e1ea1d
                job-name=testing-service
                name=testing-service
Annotations:    <none>
Parallelism:    1
Completions:    3
Start Time:     Tue, 24 Jul 2018 02:42:33 -0400
Pods Statuses:  0 Running / 3 Succeeded / 0 Failed
Pod Template:
  Labels:  controller-uid=be5f8b69-8f0c-11e8-a791-020055e1ea1d
           job-name=testing-service
           name=testing-service
  Containers:
   counter:
    Image:      centos:7
    Port:       <none>
```

```
    Host Port:  <none>
    Command:
      bin/bash
      -c
      for i in 9 8 7 6 5 4 3 2 1 ; do echo $i ; done
    Environment:  <none>
    Mounts:       <none>
  Volumes:        <none>
Events:
  Type    Reason            Age   From            Message
  ----    ------            ----  ----            -------
  Normal  SuccessfulCreate  1m    job-controller  Created pod: testing-service-strtz
  Normal  SuccessfulCreate  1m    job-controller  Created pod: testing-service-dpwr2
  Normal  SuccessfulCreate  1m    job-controller  Created pod: testing-service-7vrx5
```

We can check logs for each pod and we can see that the bash command ran as defined. If we set a for repeatable job, the pods run one after the other to perform a job defined until completion.

Parallelism - If the job does not have any dependency between them, we can have the job run in parallel. Lets create a job that lets the pods run in parallel,

```
[root@manja17-I13330 kubenetes-config]# cat basic-parallelism-job.yml
apiVersion: batch/v1
kind: Job
metadata:
  name: testing-service
spec:
  parallelism: 3
  template:
    metadata:
      labels:
        name: testing-service
    spec:
      containers:
      - name: counter
        image: centos:7
        command:
          - "bin/bash"
          - "-c"
          - "for i in 9 8 7 6 5 4 3 2 1 ; do echo $i ; done"
      restartPolicy: Never
```

```
[root@manja17-I13330 kubenetes-config]# kubectl create -f basic-parallelism-job.yml
job.batch "testing-service" created
```

```
[root@manja17-I13330 kubenetes-config]# kubectl get jobs
NAME              DESIRED   SUCCESSFUL   AGE
testing-service   <none>    2            4s
```

```
[root@manja17-I13330 kubenetes-config]# kubectl get pods
NAME                    READY     STATUS      RESTARTS   AGE
testing-service-4jq5r   0/1       Completed   0          7s
testing-service-ltlq4   0/1       Completed   0          7s
testing-service-vjn2w   0/1       Completed   0          7s
```

Describe the job using,

```
[root@manja17-I13330 kubenetes-config]# kubectl describe job testing-service
Name:           testing-service
Namespace:      default
Selector:       controller-uid=3eb65bba-8f0d-11e8-a791-020055e1ea1d
Labels:         controller-uid=3eb65bba-8f0d-11e8-a791-020055e1ea1d
                job-name=testing-service
                name=testing-service
Annotations:    <none>
Parallelism:    3
Completions:    <unset>
Start Time:     Tue, 24 Jul 2018 02:46:08 -0400
Pods Statuses:  0 Running / 3 Succeeded / 0 Failed
Pod Template:
  Labels:  controller-uid=3eb65bba-8f0d-11e8-a791-020055e1ea1d
           job-name=testing-service
           name=testing-service
  Containers:
   counter:
    Image:      centos:7
    Port:       <none>
    Host Port:  <none>
    Command:
      bin/bash
      -c
      for i in 9 8 7 6 5 4 3 2 1 ; do echo $i ; done
    Environment:  <none>
    Mounts:       <none>
  Volumes:        <none>
Events:
  Type    Reason            Age   From            Message
  ----    ------            ----  ----            -------
  Normal  SuccessfulCreate  15s   job-controller  Created pod: testing-service-vjn2w
  Normal  SuccessfulCreate  15s   job-controller  Created pod: testing-service-4jq5r
  Normal  SuccessfulCreate  15s   job-controller  Created pod: testing-service-ltlq4
```

Cron Job - Similar to Jobs kubernetes does have an option to schedule jobs. Kubernetes lets us to use cron job to perform a finite, time related tasks that run once

or repeatedly at a time that we specify. Cron jobs can be used for automatic tasks that run on specific time, such as backups , reporting and sending emails.

Cron jobs are based on jobs. they use Job objects to complete their tasks. cron Job creates a job object about once per execution time of the schedule. Cron jobs are created, managed, scaled and deleted in the same way as jobs

lets create a simple cron job using,

```
[root@manja17-I13330 kubenetes-config]# cat scheduled-jobs.yml
apiVersion: batch/v1beta1
kind: CronJob
metadata:
 name: testing-service
spec:
 schedule: "*/1 * * * *"
 jobTemplate:
   spec:
    template:
      spec:
        containers:
          - name: counter
            image: centos:7
            command:
               - "bin/bash"
               - "-c"
               - "for i in 3 2 1 ; do echo $i ; done"
        restartPolicy: OnFailure
```

The main element in the above config file is the schedule element.i am creating a pod that will be executed every 1 minute. This generally creates a pod which further creates a container called "counter" that runs a bash for loop.

The pod will be created every 1 minute and run until job is completed.

```
[root@manja17-I13330 kubenetes-config]# kubectl get jobs --watch
NAME                           DESIRED   SUCCESSFUL   AGE
testing-service-1526757000     1         0            3s
testing-service-1526757000     1         1            21s
testing-service-1526757060     1         0            0s
testing-service-1526757060     1         0            0s
testing-service-1526757060     1         1            3s
```

We can see that a pod is being created for every 1 minute which has a container "counter" running the bash for loop. We can get the cron jobs available using,

```
[root@manja17-I13330 kubenetes-config]# kubectl get cronjobs
NAME              SCHEDULE      SUSPEND   ACTIVE    LAST SCHEDULE   AGE
testing-service   */1 * * * *   False     0         50s             2m
```

Lets see that the pods are running are not, using

```
[root@manja17-I13330 kubenetes-config]# kubectl get pods
NAME                                READY     STATUS      RESTARTS   AGE
testing-service-1526757000-gh8bb    0/1       Completed   0          2m
testing-service-1526757060-2t67m    0/1       Completed   0          1m
testing-service-1526757120-z72t6    0/1       Completed   0          25s
```

Check the logs for each pod as,

```
[root@manja17-I13330 kubenetes-config]# kubectl logs
testing-service-1526757120-z72t6
3
2
1

[root@manja17-I13330 kubenetes-config]# kubectl logs
testing-service-1526757060-2t67m
3
2
1
```

Container Life Cycle Management

Life cycle management generally refers to the handling of a product as it moved through stages of pod.Kubernetes lets us to attach life cycle events to containers that are going to create. We can attach the events to containers so that they execute along with containers.

There are 2 events called postStart and preStart. K8 sends the postStart event immediately after the container is created. The postStart handler runs asynchronously relative to the Container's code, but Kubernetes' management of the container blocks until the postStart handler completes. The Container's status is not set to RUNNING until the postStart handler completes.

K8 sends the preStop event immediately before the container is terminated. the container termination will be blocked until the preStop event is completed.

Lets create a pod manifest with PreStop and postStart events as,

```
[root@manja17-I13330 kubenetes-config]# cat lifecycle-pod.yml
apiVersion: v1
kind: Pod
metadata:
 name: testing-service
spec:
 containers:
  - name: test-ser
    image: docker.io/jagadesh1982/testing-service
    ports:
    - containerPort: 9876
    lifecycle:
       postStart:
         exec:
           command: ["/bin/sh", "-c", "touch /tmp/hello"]
       preStop:
         exec:
           command: ["/bin/sh", "-c", "echo Hello from the preSop handler > /tmp/hello"]
```

In the above manifest file ,we have 2 lifecycle events defined in which postStart talks about creating a file hello in /tmp and preStop event is kicked when the container is going to terminate. The life cycle events can have similar health check events that we discussed earlier including httpGet, tcpCheck and exec.

Now lets create the pod using "kubectl" command and when we login to the pod using,

```
[root@manja17-I13330 kubenetes-config]# kubectl exec  testing-service -it -- bash
root@testing-service:/usr/src/app# ls /tmp/
hello
root@testing-service:/usr/src/app# exit
exit
```

We can see that the file is created. This file is created as a part of the postStart event which runs before the container is created.

Init Containers

In kubernetes, a Pod can have multiple containers running apps. Beside these application containers we can also have a init Containers which run before the app containers are started.

So let's take a use case, we want to have a application Container running in a Pop which has a Volume attached to the Pod. our requirement is that before the application container starts in the pod we need to have some files in our volumes so that once the application containers are up and running we can have the data ready. This is the use case where we can use init Containers.

Create a pod manifest file using,

```
[root@manja17-I13330 kubenetes-config]# cat initContainer.yml
apiVersion: v1
kind: Pod
metadata:
 name: testing-service
spec:
 containers:
   - name: test-ser
     image: docker.io/jagadesh1982/testing-service
     ports:
     - containerPort: 9876
 initContainers:
   - name:         counter
     image:        centos:7
     imagePullPolicy: IfNotPresent
     command:
        - "bin/bash"
        - "-c"
        - "for i in 9 8 7 6 5 4 3 2 1 ; do echo $i ; done"
```

The main element is the “initContianers” element which tell that a pod counter will be created from centos:7 image and will run a bash command.

Once we create the pod using the kubectl command, we can describe the pod using,

```
[root@manja17-I13330 kubenetes-config]# kubectl describe pod testing-service
```

```
Name:         testing-service
Namespace:    default
Node:         manja17-i14021/10.131.36.181
Start Time:   Mon, 21 May 2018 03:51:01 -0400
Labels:       <none>
Annotations:  <none>
Status:       Running
IP:           10.38.0.2
Init Containers:
  counter:
    Container ID:
docker://9b70db7b56681d380002666e69485b375ca707eca728675d27a3cf2bbc892226
    Image:         centos:7
    Image ID:
docker-pullable://docker.io/centos@sha256:989b936d56b1ace20ddf855a301741e52abca38
286382cba7f44443210e96d16
    Port:          <none>
    Host Port:     <none>
    Command:
      bin/bash
      -c
      for i in 9 8 7 6 5 4 3 2 1 ; do echo $i ; done
    State:         Terminated
      Reason:      Completed
      Exit Code:   0
      Started:     Mon, 21 May 2018 03:51:03 -0400
      Finished:    Mon, 21 May 2018 03:51:03 -0400
    Ready:         True
    Restart Count: 0
    Environment:   <none>
    Mounts:
      /var/run/secrets/kubernetes.io/serviceaccount from default-token-fx8mm (ro)
Containers:
  test-ser:
    Container ID:
docker://245fcadefece115124d225a2400f4d63da93ee2d59a6665c1aa3b03d55902775
    Image:         docker.io/jagadesh1982/testing-service
    Image ID:
docker-pullable://docker.io/jagadesh1982/testing-service@sha256:fa0894c592b7891c177a
```

```
22bc61eb38ca23724aa9ff9b8ea3b713b32586a75c3d
    Port:           9876/TCP
    Host Port:      0/TCP
    State:          Running
      Started:      Mon, 21 May 2018 03:51:07 -0400
    Ready:          True
    Restart Count:  0
    Environment:    <none>
    Mounts:
      /var/run/secrets/kubernetes.io/serviceaccount from default-token-fx8mm (ro)
Conditions:
  Type          Status
  Initialized   True
  Ready         True
  PodScheduled  True
Volumes:
  default-token-fx8mm:
    Type:        Secret (a volume populated by a Secret)
    SecretName:  default-token-fx8mm
    Optional:    false
QoS Class:       BestEffort
Node-Selectors:  <none>
Tolerations:     node.kubernetes.io/not-ready:NoExecute for 300s
                 node.kubernetes.io/unreachable:NoExecute for 300s
Events:
  Type    Reason                 Age   From                    Message
  ----    ------                 ----  ----                    -------
  Normal  SuccessfulMountVolume  59s   kubelet, manja17-i14021  MountVolume.SetUp
succeeded for volume "default-token-fx8mm"
  Normal  Pulled                 58s   kubelet, manja17-i14021  Container image "centos:7"
already present on machine
  Normal  Created                58s   kubelet, manja17-i14021  Created container
  Normal  Started                57s   kubelet, manja17-i14021  Started container
  Normal  Pulling                57s   kubelet, manja17-i14021  pulling image
"docker.io/jagadesh1982/testing-service"
  Normal  Pulled                 54s   kubelet, manja17-i14021  Successfully pulled image
"docker.io/jagadesh1982/testing-service"
  Normal  Created                53s   kubelet, manja17-i14021  Created container
  Normal  Started                53s   kubelet, manja17-i14021  Started container
```

Normal Scheduled 30s default-scheduler Successfully assigned testing-service to manja17-i14021

This is how we will be using a init Container.

Secrets

Passing sensitive data like username and password to a container is necessary when running in production. kubernetes defines secrets that allow us to store sensitive data such as username and password with encryption and can be passed to the pod.

A secret can be created in kubernetes using
from text file
or by creating a yml file
or by passing as Env Variables

Secret By Text File -
Creating a Secret using text file -
$ echo -n "A19fh68B001j" > ./apikey.txt

$ kubectl create secret generic apikey --from-file=./apikey.txt
secret "apikey" created

lets check the secret using ,
[root@manja17-I13330 kubenetes-config]# kubectl describe secrets/apikey
Name: apikey
Namespace: default
Labels: <none>
Annotations: <none>

Type: Opaque

Data
====
apikey.txt: 12 bytes

Create a pod manifest file by including the above secret as,
[root@manja17-I13330 kubenetes-config]# cat secret-mount-pod.yml

```
apiVersion: v1
kind: Pod
metadata:
 name: "testing-service"
 labels:
  name: "testingService"
spec:
```

```
containers:
 - image: docker.io/jagadesh1982/testing-service
   name:  "testing"
   imagePullPolicy: IfNotPresent
   ports:
     - containerPort: 9876
   volumeMounts:
     - name: apikeyvol
       mountPath: "/tmp/apikey"
       readOnly: true
volumes:
 - name: apikeyvol
   secret:
     secretName: apikey
```

In the above manifest file ,we attached the secret file by mounting the secret as a volume. the secret is mounted on the location /tmp/apikey. The mounted secret file is a read only. Create a pod using the "kubectl create -f secret-mount-pod.yml". Once the pod is created , log in to the pod and see if we can see the secretes,

```
[root@manja17-I13330 kubenetes-config]# kubectl exec testing-service -c testing -it bash
root@testing-service:/usr/src/app# mount | grep apikey
tmpfs on /tmp/apikey type tmpfs (ro,relatime,seclabel)
root@testing-service:/usr/src/app# cat /tmp/apikey/apikey.txt
A19fh68B001j
root@testing-service:/usr/src/app# exit
exit
```

From inside the container , we can see the mount point where apikey is available and when we use the cat command with the apikey.txt i can see the contents. this is how applications will also consume secrets attached to the pod.

Create a secret using the Yml file -

As we already discussed a yml file can be created using a yml file even.

```
[root@manja17-I13330 kubenetes-config]# echo 'admin' | base64
YWRtaW4K
[root@manja17-I13330 kubenetes-config]# echo 'password' | base64
cGFzc3dvcmQK
```

The above admin and password can always be decoded using,

```
[root@manja17-I13330 kubenetes-config]# echo 'YWRtaW4K' | base64 --decode
admin
```

Create a secret file as,

```
[root@manja17-I13330 kubenetes-config]# cat secrets.yml
apiVersion: v1
kind: Secret
metadata:
 name: mysecret
type: Opaque
data:
 username: YWRtaW4K
 password: cGFzc3dvcmQK
```

```
[root@manja17-I13330 kubenetes-config]# kubectl create -f secrets.yml
secret "mysecret" created
```

```
[root@manja17-I13330 kubenetes-config]# kubectl get secrets
NAME                  TYPE                                  DATA      AGE
default-token-fx8mm   kubernetes.io/service-account-token   3         2h
mysecret              Opaque                                2         5s
```

We can get the secret details using,

```
[root@manja17-I13330 kubenetes-config]# kubectl get secret mysecret -o yaml
apiVersion: v1
data:
 password: cGFzc3dvcmQK
 username: YWRtaW4K
kind: Secret
metadata:
 creationTimestamp: 2018-05-19T16:23:29Z
 name: mysecret
 namespace: default
 resourceVersion: "11094"
 selfLink: /api/v1/namespaces/default/secrets/mysecret
 uid: f7598c5a-5b80-11e8-9494-020055e1ea1d
type: Opaque
```

Create a pod with the above secret using,

```
[root@manja17-I13330 kubenetes-config]# cat secret-file-mount.yml
apiVersion: v1
kind: Pod
metadata:
 name: testing-service
spec:
 containers:
 - name: testing
   image: docker.io/jagadesh1982/testing-service
   volumeMounts:
   - name: foo
     mountPath: "/etc/foo"
     readOnly: true
 volumes:
 - name: foo
   secret:
     secretName: mysecret
```

Once the pod is created we can login to the pod and see the secrets using,

```
[root@manja17-I13330 kubenetes-config]# kubectl exec testing-service -c testing -it bash
root@testing-service:/usr/src/app# mount | grep foo
tmpfs on /etc/foo type tmpfs (ro,relatime,seclabel)
root@testing-service:/usr/src/app# cd /etc/foo
root@testing-service:/etc/foo# cat username
admin
root@testing-service:/etc/foo# cat password
password
root@testing-service:/etc/foo# exit
exit
```

Passing secret as Env Variable -

A secret when created can be also passed as environment variable to the pod. this can be done as,

```
[root@manja17-I13330 kubenetes-config]# cat secret-env-mount.yml
apiVersion: v1
kind: Pod
metadata:
```

```
 name: testing-service
spec:
 containers:
 - name: testing
   image: docker.io/jagadesh1982/testing-service
   env:
     - name: SECRET_USERNAME
       valueFrom:
         secretKeyRef:
           name: mysecret
           key: username
     - name: SECRET_PASSWORD
       valueFrom:
         secretKeyRef:
           name: mysecret
           key: password
 restartPolicy: Never
```

Iam using the same secret file created above,

```
[root@manja17-I13330 kubenetes-config]# cat secrets.yml
apiVersion: v1
kind: Secret
metadata:
 name: mysecret
type: Opaque
data:
 username: YWRtaW4K
 password: cGFzc3dvcmQK
```

Login to the pod and see if the we can see the environment variables as,

```
[root@manja17-I13330 kubenetes-config]# kubectl exec testing-service -c testing -it bash
root@testing-service:/usr/src/app# set | grep SECRET
SECRET_PASSWORD=$'password\n'
SECRET_USERNAME=$'admin\n'
root@testing-service:/usr/src/app# exit
exit
```

Persistent Volumes

There will be always cases where we need to save data on to a drive or disk. Pods when stopped will destroy the data stored inside the container. Node reboots also clear any data from the Ram based disks. There are many times where we need some shared spacer have containers that process data and hand it off to another container before they die.

Kubernetes provides us a way to manage storage for containers. A Volume is a directory accessible to all running containers in the pod, with a guarantee that the data is preserved. In this article we will see how we can create volumes and attach them to the pods. For the demo we will use NFS as our additional drive.

Depending the type of volumes, we differentiate the types of volumes as,

1. *Node-local* volumes, such as emptyDir or hostPath
2. Generic *networked* volumes, such as nfs, glusterfs, or cephfs
3. Cloud provider–specific volumes, such as awsElasticBlockStore, azureDisk, or gcePersistentDisk
4. Special-purpose volumes, such as secret or gitRepo

Empty directory - One of the easiest ways to achieve improved persistence amid container crashes and data sharing within a pod is to use the emptydir volume. This volume type can be used with either the storage volumes of the node machine itself or an optional RAM disk for higher performance.

emptyDir volume is created when the pod is assigned to the node and exists until the pod runs on that node.As the name says , the volume is initially empty and pod can read and write contents to the volume.

Volume with Directory Type -

```
[root@manja17-I13330 kubenetes-config]# cat emptyDir-mysql.yml
apiVersion: v1
kind: Pod
metadata:
  name: "mysql"
  labels:
   name: "lbl-k8s-mysql"
spec:
  containers:
   - image: docker.io/jagadesh1982/testing-service
     name:  "testing"
     imagePullPolicy: IfNotPresent
     ports:
      - containerPort: 3306
     volumeMounts:
```

```
      - mountPath: /data-mount
        name: data
  volumes:
    - name: data
      emptyDir: {}
```

Login to the container and see how it goes,

```
[root@manja17-I13330 kubenetes-config]# kubectl exec mysql -c testing -it bash
root@mysql:/usr/src/app# cd /data-mount
root@mysql:/data-mount# pwd
/data-mount
root@mysql:/data-mount# touch hello
root@mysql:/data-mount# echo "hello world" >> ./hello
root@mysql:/data-mount# exit
exit
```

Volume with Memory Type - By default emptyDir volumes are stored on whatever medium is backing the node. It that is an SSD disk the node restarts will still have the data available. There is another type of backend "memory" which tells k8 to mount a tmpfs (Ram backed file System). While this is fast , any node restart clears the contents.

Lets create a Volume type with

```
[root@manja17-I13330 kubenetes-config]# cat emptyDir-memory-pod.yml
apiVersion: v1
kind: Pod
metadata:
  name: "mysql"
  labels:
   name: "lbl-k8s-mysql"
spec:
  containers:
   - image: docker.io/jagadesh1982/testing-service
     name:  "testing"
     imagePullPolicy: IfNotPresent
     ports:
      - containerPort: 3306
     volumeMounts:
       - mountPath: /data-mount
         name: data
  volumes:
    - name: data
      emptyDir:
        medium: Memory
```

```
[root@manja17-I13330 kubenetes-config]# kubectl exec mysql -c testing -it bash
root@mysql:/# cd data-mount/
root@mysql:/data-mount# pwd
/data-mount
```

root@mysql:/# exit
Exit

Volume Type as NFS or any Cloud Backed Storage - While making a NFS or any cloud backed storage we Will require some additional components. We need to create a Persistent Volume and Persistent volume Claim for using the volumes

PersistentVolume - Low level representation of the storage volume
PersistentVolumeClaim - Binding between a Pod and Persistent Volume
Volume Driver - code that will be used to communicate with the back end storage
StorageClass - Allows dynamic provisioning of Persistent volume

Lets create a Persistent Volume,

```
[root@manja17-I13330 kubenetes-config]# cat nfs-pv.yml
apiVersion: v1
kind: PersistentVolume
metadata:
  name: test-nfs
spec:
  capacity:
    storage: 1Mi
  accessModes:
    - ReadWriteMany
  nfs:
    server: 10.131.175.138
    path: "/nfsshare"
```

In the above config, we are actually using a nfs volume from the ip address "10.131.175.138" where nfs shares where already created under the /nfsshare. Run this and see how it works

```
[root@manja17-I13330 kubenetes-config]# kubectl create -f nfs-pv.yml
persistentvolume "test-nfs" created
```

```
[root@manja17-I13330 kubenetes-config]# kubectl get pv
NAME       CAPACITY   ACCESS MODES   RECLAIM POLICY   STATUS    CLAIM             STORAGECLASS   REASON    AGE
test-nfs   1Mi        RWX            Retain           Bound     default/test-nfs                           4h
```

If we check the mount command from our worker nodes we can see the below output,

```
[root@manja17-I14021 nfsshare]# mount | grep nfsshare
10.131.175.138:/nfsshare on /mnt/nfsshare type nfs4
(rw,relatime,vers=4.1,rsize=1048576,wsize=1048576,namlen=255,hard,proto=tcp,port=0,
timeo=600,retrans=2,sec=sys,clientaddr=10.131.36.181,local_lock=none,addr=10.131.17
5.138)
```

We can see that the /nfsshare from 10.131.165.138 is mounted onto the /mnt/nfssshare on the worker nodes. In the above case we are creating a persistent volume with 1MI size

Now lets create a PersistentVolumeClaim -
[root@manja17-I13330 kubenetes-config]# cat nfs-pvc.yml

```
kind: PersistentVolumeClaim
apiVersion: v1
metadata:
  name: test-nfs
spec:
  accessModes:
    - ReadWriteMany
  resources:
    requests:
      storage: 1Mi
```

The Persistent volume claim works in different way. In the above configuration we are asking K8 to provide us a volume which has storage of 1MI. We are not calling the persistent volume directly rather we are creating a persistent Volume claim and giving K8 the authority of identifying the volumes with the size that we requested.if that finds with a pv with that volume it allocates that to the pod.

```
[root@manja17-I13330 kubenetes-config]# kubectl get pvc
NAME       STATUS    VOLUME     CAPACITY   ACCESS MODES   STORAGECLASS   AGE
test-nfs   Bound     test-nfs   1Mi        RWX                           4h
```

Sample Pod Configuration - Now lets create a Pod which will use the NFS volume
[root@manja17-I13330 kubenetes-config]# cat nfs-pod.yml

```
apiVersion: v1
kind: ReplicationController
metadata:
  name: nfs-web
spec:
  replicas: 1
  selector:
    role: web-frontend
  template:
    metadata:
      labels:
        role: web-frontend
    spec:
      containers:
      - name: web
        image: nginx
        ports:
          - name: web
            containerPort: 80
        volumeMounts:
            # name must match the volume name below
            - name: test-nfs
              mountPath: "/usr/share/nginx/html"
```

```
  volumes:
  - name: test-nfs
    persistentVolumeClaim:
      claimName: test-nfs
```

In the above pod configuration, the main elements are Volumes and Volume mounts. The volume element contains the Persistent Volume claim details that we created earlier. The Volume mount is the place where if volume with our requirement is available it will mount on the location /usr/share/nginx/html inside the container.

Run the Container and see how it works -

```
[root@manja17-I13330 kubenetes-config]# kubectl get pods
NAME                    READY         STATUS            RESTARTS   AGE
nfs-web-qk2qn          1/1             Running          0              4m

[root@manja17-I13330 kubenetes-config]# kubectl exec -it nfs-web-qk2qn -- bash
root@nfs-web-qk2qn:/# cd /usr/share/nginx/html/
root@nfs-web-qk2qn:/usr/share/nginx/html# touch container.txt
root@nfs-web-qk2qn:/usr/share/nginx/html# cat container.txt
root@nfs-web-qk2qn:/usr/share/nginx/html# echo "this is from container" >> container.txt
root@nfs-web-qk2qn:/usr/share/nginx/html# cat container.txt
this is from container
root@nfs-web-qk2qn:/usr/share/nginx/html# exit
exit
```

Config Maps

While working with application running inside containers, there will be always be a need for the config file. Application that need to connect to backends and other config values are always sent as a configuration files. When we have our application in a container , the dependencies and configurations are also need to be added as a part of image.

Kubernetes provides us a different way of passing configuration by using configMaps. Configmap is one of two ways to provide configurations to your application. ConfigMaps is a simple key/value store, which can store simple values to files.

ConfigMaps is a way to decouple the application-specific artifacts from the container image, thereby enabling better portability and externalization. lets create a configMap with the configuration data,

```
[root@manja17-I13330 kubenetes-config]# cat configMap.yml
apiVersion: v1
kind: ConfigMap
metadata:
 name: testing-service-staging
 labels:
   name: testing-service-staging
data:
 config: |-
   ---
   :verbose: true
   :environment: test
   :logfile: log/sidekiq.log
   :concurrency: 20
   :queues:
     - [default, 1]
   :dynamic: true
   :timeout: 300
```

I created a config Map which includes my configuration for the application.

```
[root@manja17-I13330 kubenetes-config]# kubectl create -f configMap.yml
configmap "testing-service-staging" created

[root@manja17-I13330 kubenetes-config]# kubectl get configmaps
NAME                      DATA      AGE
testing-service-staging   1         4s
```

Now create a pod with the config map as,

```
[root@manja17-I13330 kubenetes-config]# cat configmap-file-mount.yml
apiVersion: v1
kind: Pod
metadata:
 name: testing-service
spec:
 containers:
 - name: testing
   image: docker.io/jagadesh1982/testing-service
   volumeMounts:
     - mountPath: /etc/sidekiq/config
       name: testing-service-staging
 volumes:
   - name: testing-service-staging
     configMap:
        name: testing-service-staging
        items:
         - key: config
           path:  testing-stage.yml
```

We can see that the config map is even mounted as a volume. In the above case we created a config Map with the one that we created earlier as"testing-service-staging" and then mounted it on /etc/sidekiq/config. the file that we created is testing-stage.yml with the config data available in the config map that we created earlier.

Now access the configuration file that we mounted using as,

```
[root@manja17-I13330 kubenetes-config]# kubectl exec testing-service -c testing -it bash
root@testing-service:/usr/src/app# cd /etc/sidekiq/
root@testing-service:/etc/sidekiq# ls
config
root@testing-service:/etc/sidekiq# cat config/
cat: config/: Is a directory
root@testing-service:/etc/sidekiq# cd config/
root@testing-service:/etc/sidekiq/config# ls
testing-stage.yml
```

```
root@testing-service:/etc/sidekiq/config# cat testing-stage.yml
---
:verbose: true
:environment: test
:logfile: log/sidekiq.log
:concurrency: 20
:queues:
 - [default, 1]
:dynamic: true
:timeout: 300root@testing-service:/etc/sidekiq/config# exit
exit
```

We can see that when we access the config file testing-stage.yml file we have the same config data from the configmap that we created.

Image Pull Secrets

Most times we will be having our container images created and uploaded to the internal company docker registry. for accessing these private images , we need to provide username and password or atleast a secret token to access them. These usernames and password are used during the image pull from the repository. In this article we will see how we can make a image private in a docker hub and use a secret to download the images.

In Kubernetes we use the secret of docker-registry type to authenticate with a container registry to pull a private image

One of the image in docker hub is made a private. when we tried to access the image,

```
[root@manja17-I13330 kubenetes-config]# docker pull
docker.io/jagadesh1982/testing-service
Using default tag: latest
Trying to pull repository docker.io/jagadesh1982/testing-service ...

Get https://registry-1.docker.io/v2/jagadesh1982/testing-service/manifests/latest:
unauthorized: incorrect username or password
```

lets create a secret for accessing the docker hub in kubernetes using,

```
[root@manja17-I13330 kubenetes-config]# kubectl create secret docker-registry
docker-secret --docker-server https://index.docker.io/v1/ --docker-email
jagadesh.manchala@gmail.com --docker-username=<user Name> --docker-password
<Password>
secret "docker-secret" created

[root@manja17-I13330 kubenetes-config]# kubectl get secrets
NAME                  TYPE                                  DATA      AGE
default-token-fx8mm   kubernetes.io/service-account-token   3         2h
docker-secret         kubernetes.io/dockerconfigjson        1         9s

[root@manja17-I13330 kubenetes-config]# kubectl describe secret docker-secret
Name:         docker-secret
Namespace:    default
Labels:       <none>
Annotations:  <none>

Type:  kubernetes.io/dockerconfigjson
```

```
Data
====
.dockerconfigjson:  180 bytes
```

Configure the Pod with the secret to pull the images as,

```
[root@manja17-I13330 kubenetes-config]# cat image-pull-secret.yml
apiVersion: v1
kind: Pod
metadata:
 name: testing-service
spec:
 containers:
   - name: test-ser
     image: docker.io/jagadesh1982/testing-service
     ports:
      - containerPort: 9876
 imagePullSecrets:
   - name: docker-secret
```

Now lets check the pod using,

```
[root@manja17-I13330 kubenetes-config]# kubectl get pods
NAME              READY     STATUS    RESTARTS   AGE
testing-service   1/1       Running     0          43s
```

We have defined the secret that we created to access the docker hub for downloading images in the pod manifest file. Since we defined the secret to download them , we can successfully create the pod and use.

Deployment Strategies

Release management is very important while playing with containers. It is very necessary to plan the release management for containers based on the application and infrastructure. Kubernetes provides multiple ways of release management or we can call deployment strategies. Kubernetes has a Strategy element that we can define in our deployment manifest file to define how the deployment works.

Release Strategy – Recreate

The recreate is quite simple. When we define a deployment manifest file with Recreate strategy, all running application pods will be terminated and new application version pods will be created.

Pros: application state is entirely new
Cons: There will be an outage for the application

Let's see an example of how to implement the recreate strategy,

```
[root@manja17-I13330 kubenetes-config]# cat recreate-deployment-v1.yml
apiVersion: v1
kind: Service
metadata:
  name: testing-service
  labels:
        app: testings-service
spec:
  type: NodePort
  ports:
  - name: http
        port: 80
        targetPort: http
  selector:
        app: testing-service
---
apiVersion: apps/v1beta1
kind: Deployment
metadata:
  name: testing-service
  labels:
        app: testing-service
```

```
spec:
  replicas: 3
  selector:
        matchLabels:
        app: testing-service
  strategy:
        type: Recreate
  selector:
        matchLabels:
        app: testing-service
  template:
        metadata:
        labels:
        app: testing-service
        version: "1.0"
        spec:
        containers:
        - name: test-service
        image: docker.io/jagadesh1982/testing-service
        ports:
        - name: http
         containerPort: 9876
        env:
        - name: VERSION
        value: "1.0"
```

In the above deployment manifest file, I defined an service and also a deployment. The example is quite east to understand. The important element in the above configuration is,

```
strategy:
        type: Recreate
```

Run the deployment using,

```
[root@manja17-I13330 kubenetes-config]# kubectl create -f recreate-deployment-v1.yml
service "testing-service" created
deployment.apps "testing-service" created
```

Once done check the objects that are created from the above command using,

```
[root@manja17-I13330 kubenetes-config]# kubectl get all -l app=testing-service
NAME                                    READY   STATUS    RESTARTS   AGE
pod/testing-service-d48f8c647-2gw9b     1/1     Running   0          18s
pod/testing-service-d48f8c647-59mx2     1/1     Running   0          18s
pod/testing-service-d48f8c647-z682m     1/1     Running   0          18s

NAME                                DESIRED   CURRENT   UP-TO-DATE   AVAILABLE   AGE
deployment.apps/testing-service     3         3         3            3           18s

NAME                                          DESIRED   CURRENT   READY   AGE
replicaset.apps/testing-service-d48f8c647     3         3         3       18s
```

We can see that the deployment is created as well as a service also. We can access the Application using,

```
[root@manja17-I13330 kubenetes-config]# curl 10.109.124.147:80/info
{"host": "10.109.124.147", "version": "1.0", "from": "10.32.0.1"}
```

We can see the version from the output is 1. Now lets deploy a version 2 using the recreate deployment strategy,

```
[root@manja17-I13330 kubenetes-config]# cat recreate-deployment-v2.yml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: testing-service
  labels:
        app: testing-service
spec:
  replicas: 3
  strategy:
        type: Recreate
  selector:
        matchLabels:
        app: testing-service
  template:
        metadata:
        labels:
        app: testing-service
        version: "2.0"
        spec:
```

```
containers:
- name: test-service
image: docker.io/jagadesh1982/testing-service
ports:
- name: http
containerPort: 9876
env:
- name: VERSION
value: "2.0"
```

In the above configuration file, we are just updating the deployment file. We changed the version of the application to 2.

Now if we run this deployment file using,

```
[root@manja17-I13330 kubenetes-config]# kubectl apply -f recreate-deployment-v2.yml
Warning: kubectl apply should be used on resource created by either kubectl create --save-config or kubectl apply
deployment.apps "testing-service" configured

[root@manja17-I13330 kubenetes-config]# kubectl get pods --watch
NAME                                READY   STATUS              RESTARTS   AGE
testing-service-d48f8c647-2gw9b     1/1     Terminating         0          8m
testing-service-d48f8c647-59mx2     1/1     Terminating         0          8m
testing-service-d48f8c647-z682m     1/1     Terminating         0          8m
testing-service-5c87dc5b6c-llkjm    0/1       Pending             0            0s
testing-service-5c87dc5b6c-2srx6    0/1     Pending             0          0s
testing-service-5c87dc5b6c-hdpz5    0/1     Pending             0          0s
testing-service-5c87dc5b6c-llkjm    0/1     ContainerCreating   0          0s
testing-service-5c87dc5b6c-hdpz5    0/1     ContainerCreating   0          0s
testing-service-5c87dc5b6c-2srx6    0/1     ContainerCreating   0          0s
testing-service-5c87dc5b6c-2srx6    1/1     Running             0          5s
testing-service-5c87dc5b6c-llkjm    1/1     Running             0          5s
testing-service-5c87dc5b6c-hdpz5    1/1     Running             0          8s
```

If we watch the pods, we can see that the older version pods will be terminated, and the newer version pods are created. If we access the application, we can see

```
[root@manja17-I13330 kubenetes-config]# curl 10.109.124.147:80/info
{"host": "10.109.124.147", "version": "2.0", "from": "10.32.0.1"}
```

We can see the version is 2 over here. **So how does this happen?**
The selector field is the main one which does the magic. The selector field tell the deployment which pod to update with the newer version. This will define how the deployment finds which pod to manage.

Delete all the objects that we created using,

```
[root@manja17-I13330 kubenetes-config]#  kubectl delete all -l app=testing-service
pod "testing-service-5c87dc5b6c-2srx6" deleted
pod "testing-service-5c87dc5b6c-hdpz5" deleted
pod "testing-service-5c87dc5b6c-llkjm" deleted
deployment.apps "testing-service" deleted
```

Deployment Strategy - Rolling Update
In the next series of K8 release management, we will see how rolling update works. A rolling deployment updates the pod in a rolling update where a secondary set of pods will be created while the first set are terminated. So while the first set is terminated slowly one by one, second version of pods are created one by one.

If we trigger a deployment while an existing rollout is in progress, the deployment will pause the rollout and proceed to the new release by overriding the rollout. So be cautious.

Pros :
Version is slowly released across worker nodes

Cons:
Rollout/rollback can take more time
Traffic to the pods needs to be handled correctly.

Lets see an example of how rolling deployment works. The same example will be used,

```
[root@manja17-I13330 kubenetes-config]# cat rolling-update-deployment-v1.yml
apiVersion: v1
kind: Service
metadata:
  name: testing-service
  labels:
        app: testing-service
spec:
  type: NodePort
  ports:
  - name: http
        port: 80
        targetPort: http
  selector:
        app: testing-service
---
```

```
apiVersion: apps/v1beta1
kind: Deployment
metadata:
  name: testing-service
  labels:
        app: testing-service
spec:
  replicas: 10
  selector:
        matchLabels:
        app: testing-service
  template:
        metadata:
        labels:
        app: testing-service
        version: "1.0"
        spec:
        containers:
        - name: test-service
        image: docker.io/jagadesh1982/testing-service
        ports:
        - name: http
        containerPort: 9876
        env:
        - name: VERSION
        value: "1.0"
```

Lets create the deployment using,

```
[root@manja17-I13330 kubenetes-config]# kubectl create -f rolling-update-deployment.yml
service "testing-service" created
deployment.apps "testing-service" created
```

Once the deployment is created, we can see 10 pods being created. A service as well as the deployment with 10 pods are created. We can use the curl command to access the application as,

```
[root@manja17-I13330 kubenetes-config]# curl 10.100.196.68/info
{"host": "10.100.196.68", "version": "1.0", "from": "10.32.0.1"}
```

Now lets deploy the second version of the application using rolling update. The manifest file for the second deployment looks as,

```
[root@manja17-I13330 kubenetes-config]# cat rolling-update-deployment-v2.yml
apiVersion: apps/v1beta1
kind: Deployment
metadata:
  name: testing-service
  labels:
        app: testing-service
```

```
spec:
  replicas: 10
  strategy:
        type: RollingUpdate
        rollingUpdate:
        maxSurge: 1
        maxUnavailable: 0
  selector:
        matchLabels:
        app: testing-service
  template:
        metadata:
        labels:
        app: testing-service
        version: "2.0"
        spec:
        containers:
        - name: test-service
        image: docker.io/jagadesh1982/testing-service
        ports:
        - name: http
            containerPort: 9876
        env:
        - name: VERSION
        value: "2.0"
```

The important element in the above configuration file is,

```
strategy:
        type: RollingUpdate
        rollingUpdate:
        maxSurge: 1
        maxUnavailable: 0
```

We have defined the RollingUpdate as a deployment strategy. We have to define the maxSurge and maxUnavailable elements for the Rolling Update.
maxSurge – defines how many pods we can add at a time
maxUnavailable – defines how many pods can be unavailable

The maxUnavailable parameter is the maximum number of pods that can be unavailable during the update. The maxSurge parameter is the maximum number of pods that can be scheduled above the original number of pods. Both parameters can be set to either a percentage (e.g., 10%) or an absolute value (e.g., 2). The default value for both is 25%

```
[root@manja17-I13330 kubenetes-config]# kubectl apply -f
rolling-update-deployment-v2.yml
Warning: kubectl apply should be used on resource created by either kubectl create
--save-config or kubectl apply
```

deployment.apps "testing-service" configured

Also watch the pods using “watch kubectl get po”.We will see how the first set of pods will be terminated and the second version of the application pod are up and running.

Lets try to access the service using,
[root@manja17-I13330 kubenetes-config]# while sleep 0.1; do curl "10.100.196.68/info"; done
{"host": "10.100.196.68", "version": "2.0", "from": "10.32.0.1"}{"host": "10.100.196.68", "version": "2.0", "from": "10.32.0.1"}{"host": "10.100.196.68", "version": "1.0", "from": "10.32.0.1"}{"host": "10.100.196.68", "version": "1.0", "from": "10.32.0.1"}{"host": "10.100.196.68", "version": "2.0", "from": "10.32.0.1"}{"host": "10.100.196.68", "version": "1.0", "from": "10.32.0.1"}{"host": "10.100.196.68", "version": "1.0", "from": "10.32.0.1"}{"host": "10.100.196.68", "version": "1.0", "from": "10.32.0.1"}{"host": "10.100.196.68", "version": "1.0", "from": "10.32.0.1"}{"host": "10.100.196.68", "version": "1.0", "from": "10.32.0.1"}{"host": "10.100.196.68", "version": "1.0", "from": "10.32.0.1"}{"host": "10.100.196.68", "version": "1.0", "from": "10.32.0.1"}{"host": "10.100.196.68", "version": "2.0", "from": "10.32.0.1"}{"host": "10.100.196.68", "version": "2.0", "from": "10.32.0.1"}{"host": "10.100.196.68", "version": "1.0", "from": "10.32.0.1"}{"host": "10.100.196.68", "version": "1.0", "from": "10.32.0.1"}{"host": "10.100.196.68", "version": "1.0", "from": "10.32.0.1"}{"host": "10.100.196.68", "version": "2.0", "from": "10.32.0.1"}{"host": "10.100.196.68", "version": "1.0", "from": "10.32.0.1"}{"host": "10.100.196.68", "version": "2.0", "from": "10.32.0.1"}

In the above snippet, I just used a while loop for hitting the service for a response from the application. We can see some hit are returning version as 1 and some are returning version as 2. We need to have a fine control over the traffic.

Roll back a Deployment - In case we find some issues and want to rollback to a older version we can do some thing like this

[root@manja17-I13330 kubenetes-config]# kubectl rollout undo deploy testing-service
[root@manja17-I13330 kubenetes-config]# curl 10.100.196.68/info
{"host": "10.100.196.68", "version": "1.0", "from": "10.32.0.1"}

Pause a Deployment - If you can also pause the rollout if you want to run the application for a subset of users:
[root@manja17-I13330 kubenetes-config]# Kubectl rollout pause deploy testing-service

Happy with the Deployment- Once we are happy with the newer version we can rollout the rest using,
[root@manja17-I13330 kubenetes-config]# Kubectl rollout resume deploy testing-service

This is how the rolling update works.

Deployment Strategy - Blue/Green

In this 3 post we will see how to do a blue/green deployment. A blue/green deployment is different from the rolling update in which a green version of the application is deployed along with the blue version. After testing the green version and once satisfied we will update the service to point to the green version. This way we will route the traffic from blue version to green version. This is done by playing with the labels in the selector fields. Once the traffic is moved to the green version, the blue version pods are terminated.

Lets create the blue deployment first as,

```
[root@manja17-I13330 kubenetes-config]# cat  blue-deployment.yml
apiVersion: v1
kind: Service
metadata:
  name: testing-service
  labels:
    app: testing-service
spec:
  type: NodePort
  ports:
  - name: http
    port: 80
    targetPort: http

  # Note here that we match both the app and the version
  selector:
    app: testing-service
    version: "1.0"
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: testing-service-v1
  labels:
    app: testing-service
spec:
  replicas: 3
  selector:
    matchLabels:
      app: testing-service
      version: "1.0"
  template:
    metadata:
      labels:
        app: testing-service
        version: "1.0"
    spec:
      containers:
      - name: testing-service
```

```
      image: docker.io/jagadesh1982/testing-service
      ports:
      - name: http
        containerPort: 9876
      env:
      - name: VERSION
        value: "1.0"
```

```
[root@manja17-I13330 kubenetes-config]# kubectl apply -f blue-deployment.yml
service "testing-service" created
deployment.apps "testing-service-v1" created

[root@manja17-I13330 kubenetes-config]# kubectl get pods
NAME                                 READY     STATUS  RESTARTS   AGE
testing-service-v1-bc6866889-g8zrr   1/1       Running      0          5m
testing-service-v1-bc6866889-lj8f7   1/1       Running      0          5m
testing-service-v1-bc6866889-qnr2r   1/1       Running      0          5m

[root@manja17-I13330 kubenetes-config]# kubectl get svc
NAME              TYPE        CLUSTER-IP          EXTERNAL-IP   PORT(S)         AGE
kubernetes    ClusterIP    10.96.0.1         <none>        443/TCP         1d
testing-service   NodePort    10.110.200.110  <none>        80:30009/TCP   5m

[root@manja17-I13330 kubenetes-config]# curl 10.99.143.66/info
{"host": "10.99.143.66", "version": "1.0", "from": "10.32.0.1"}
```

Lets run the green deployment,with the below manifest file,

```
[root@manja17-I13330 kubenetes-config]# cat green-deployment.yml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: testing-service-v2
  labels:
    app: testing-service
spec:
  replicas: 3
  selector:
    matchLabels:
      app: testing-service
      version: "2.0"
  template:
    metadata:
      labels:
        app: testing-service
        version: "2.0"
    spec:
      containers:
      - name: testing-service
```

```
      image: docker.io/jagadesh1982/testing-service
      ports:
      - name: http
        containerPort: 9876
      env:
      - name: VERSION
        value: "2.0"
```

```
[root@manja17-I13330 kubenetes-config]# kubectl apply -f green-deployment.yml
deployment.apps "testing-service-v2" created

[root@manja17-I13330 kubenetes-config]# kubectl get pods
NAME                                  READY     STATUS        RESTARTS   AGE
testing-service-v1-bc6866889-g8zrr    1/1       Running       0          8m
testing-service-v1-bc6866889-lj8f7    1/1       Running       0          8m
testing-service-v1-bc6866889-qnr2r    1/1       Running       0          8m
testing-service-v2-7f966d5d4c-mb2fk   1/1       Running       0          2m
testing-service-v2-7f966d5d4c-qs6fz   1/1       Running       0          2m
testing-service-v2-7f966d5d4c-z4tj4   1/1       Running       0          2m
```

Side by side, 3 pods are running with version 2 but the service still send traffic to the first deployment. If necessary, you can manually test one of the pod by port-forwarding it to your local environment. Once your are ready, you can switch the traffic to the new version by patching the service to send traffic to all pods with label version=v2.0.0:

```
[root@manja17-I13330 kubenetes-config]# kubectl patch service testing-service -p '{"spec":{"selector":{"version":"2.0"}}}'
service "testing-service" patched

[root@manja17-I13330 kubenetes-config]# kubectl get svc
NAME              TYPE          CLUSTER-IP      EXTERNAL-IP   PORT(S)          AGE
kubernetes        ClusterIP     10.96.0.1       <none>        443/TCP          1d
testing-service   NodePort      10.99.143.66    <none>        80:30126/TCP     3m

[root@manja17-I13330 kubenetes-config]# curl 10.99.143.66/info
{"host": "10.99.143.66", "version": "2.0", "from": "10.32.0.1"}
```

If you want to rollback to the older version , we can use

```
[root@manja17-I13330 kubenetes-config]# kubectl patch service testing-service -p '{"spec":{"selector":{"version":"1.0"}}}'
service "testing-service" patched

[root@manja17-I13330 kubenetes-config]# kubectl get svc
NAME              TYPE          CLUSTER-IP      EXTERNAL-IP   PORT(S)          AGE
kubernetes        ClusterIP     10.96.0.1       <none>        443/TCP          1d
testing-service   NodePort      10.99.143.66    <none>        80:30126/TCP     4m
```

```
[root@manja17-I13330 kubenetes-config]# curl 10.99.143.66/info
{"host": "10.99.143.66", "version": "1.0", "from": "10.32.0.1"}

[root@manja17-I13330 kubenetes-config]# kubectl get deploy
NAME               DESIRED   CURRENT   UP-TO-DATE   AVAILABLE   AGE
testing-service-v1   3         3         3            3           5m
testing-service-v2   3         3         3            3           2m
```

Once we are happy with the green version, delete the blue deployment using,

```
[root@manja17-I13330 kubenetes-config]# kubectl delete deploy testing-service-v1
deployment.extensions "testing-service-v1" deleted
```

Now check the pods ,

```
[root@manja17-I13330 kubenetes-config]# kubectl get pods
NAME                                 READY   STATUS        RESTARTS   AGE
testing-service-v1-bc6866889-dthrt   1/1     Terminating   0          5m
testing-service-v1-bc6866889-nk282   1/1     Terminating   0          5m
testing-service-v1-bc6866889-rxdtz   1/1     Terminating   0          5m
testing-service-v2-76dcdbd6bc-g56bh  1/1     Running       0          3m
testing-service-v2-76dcdbd6bc-jbvkw  1/1     Running       0          3m
testing-service-v2-76dcdbd6bc-s4zg2  1/1     Running       0          3m
```

We can see the blue version of pods are being terminated and green version are up and running. This is how we will be doing a blue/green deployment in kubernetes.

Horizontal Pod Accelerator

Yet to Come

Ingress Controller

Yet to Come

Helm Package Management

Yet To Come

Cheat Sheet

List all Container Images available

```
[root@manja17-I13330 kubenetes-config]# kubectl get pods --all-namespaces -o
jsonpath="{..image}" |\
> tr -s '[[:space:]]' '\n' |\
> sort |\
> uniq -c
      1 docker.io/rajpr01/myapp
      1 docker.io/rajpr01/myapp:latest
      3 docker.io/weaveworks/weave-kube:2.3.0
      3 docker.io/weaveworks/weave-npc:2.3.0
      2 k8s.gcr.io/etcd-amd64:3.1.12
      2 k8s.gcr.io/k8s-dns-dnsmasq-nanny-amd64:1.14.8
      2 k8s.gcr.io/k8s-dns-kube-dns-amd64:1.14.8
      2 k8s.gcr.io/k8s-dns-sidecar-amd64:1.14.8
      2 k8s.gcr.io/kube-apiserver-amd64:v1.10.2
      2 k8s.gcr.io/kube-controller-manager-amd64:v1.10.2
      6 k8s.gcr.io/kube-proxy-amd64:v1.10.2
      2 k8s.gcr.io/kube-scheduler-amd64:v1.10.2
      3 weaveworks/weave-kube:2.3.0
      3 weaveworks/weave-npc:2.3.0
```

List Containers by POD

```
[root@manja17-I13330 kubenetes-config]# kubectl get pods --all-namespaces
-o=jsonpath='{range .items[*]}{"\n"}{.metadata.name}{":\t"}{range
.spec.containers[*]}{.image}{", "}{end}{end}' |sort

etcd-manja17-i13330:    k8s.gcr.io/etcd-amd64:3.1.12,
kube-apiserver-manja17-i13330:  k8s.gcr.io/kube-apiserver-amd64:v1.10.2,
********
```