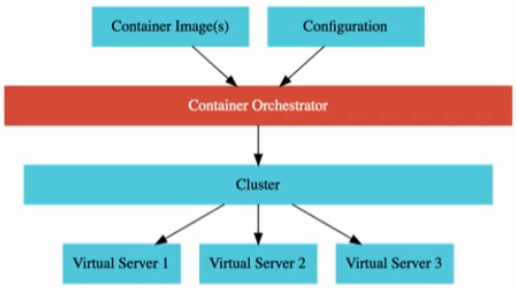
Kubernetes (K8S) with Microservices using Docker, Spring Boot, Spring Cloud

- We are able to create containers for all our Microservices & run them using Docker compose. However, there are a lot more requirements to talk about when we want to run microservices in production.
- **For e.g.,** (Multiple instances of microservice containers, Auto scaling, Service Discovery, Load balancing, self-healing, Zero downtime deployments).

Container Orchestration

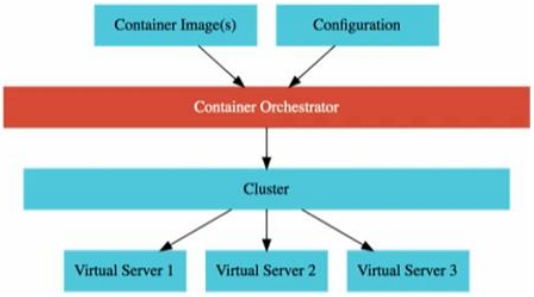
- **Requirement** : I want 10 instances of Microservice A container, 15 instances of Microservice B container and ....
- Typical Features:
  - **Auto Scaling** - Scale containers based on demand
  - **Service Discovery** - Help microservices find one another
  - **Load Balancer** - Distribute load among multiple instances of a microservice
  - **Self Healing** - Do health checks and replace failing instances
  - **Zero Downtime Deployments** - Release new versions without downtime



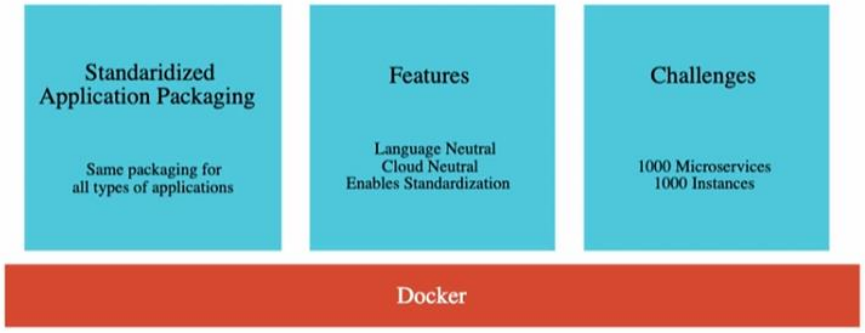
- These are the features which the container orchestration tools provide us with & the **popular container orchestration tools** present are:

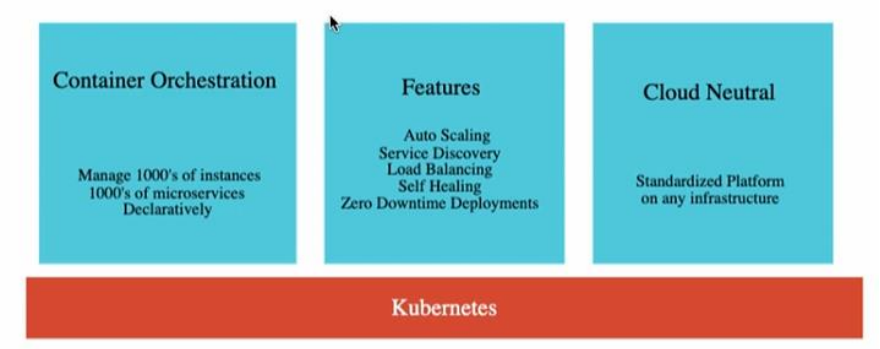
Container Orchestration Options

- **AWS Specific**
  - AWS Elastic Container Service (ECS)
  - AWS Fargate : Serverless version of AWS ECS
- **Cloud Neutral** - Kubernetes
  - AWS - Elastic Kubernetes Service (EKS)
  - Azure - Azure Kubernetes Service (AKS)
  - GCP - Google Kubernetes Engine (GKE)
  - EKS/AKS does not have a free tier!
    - We use GCP and GKE!

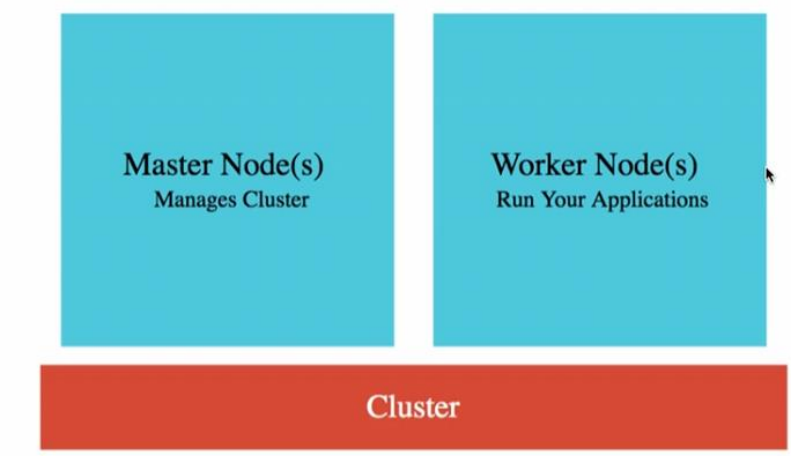


Q. Why do we need Docker
Ans: Docker enable standardization of our package & deploy our application irrespective of the language or framework used to build the application or the platform where we'd want to deploy the application to.





Kubernetes (K8S) Architecture



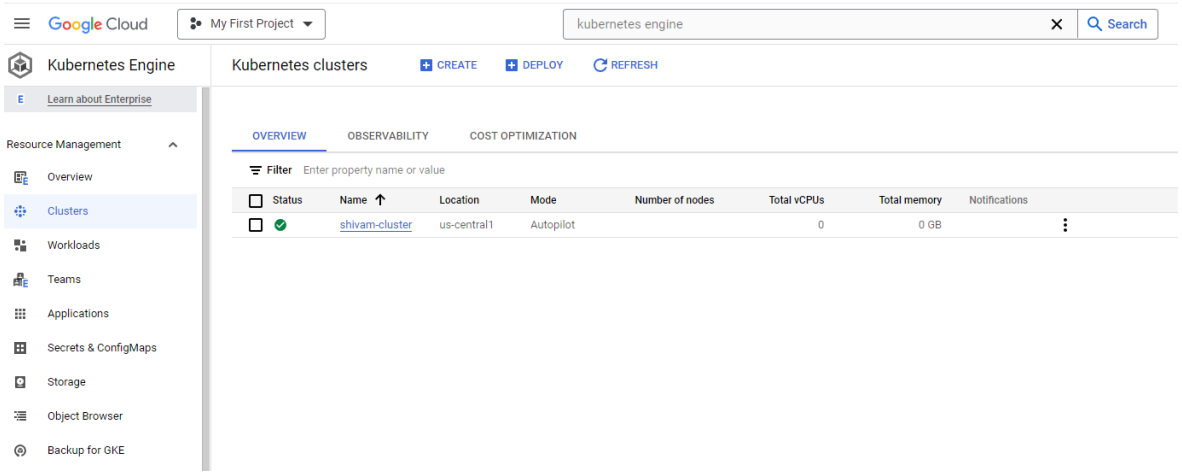
- Kubernetes is the best resource manager ever & the resources here are virtual servers (servers in cloud) called as Nodes (in Kubernetes). Kubernetes can manage 1000s of such nodes.
- To manage thousands of Kubernetes nodes, we have a few master nodes. Typically, we'll have one master node but when we need high availability, we go for multiple master nodes.
- The cluster is the combination of worker nodes & master nodes.
  - The nodes that do the work are called worker nodes.
  - The nodes that do the management work are called master nodes. Master nodes ensure that the nodes are available & are doing some useful work.

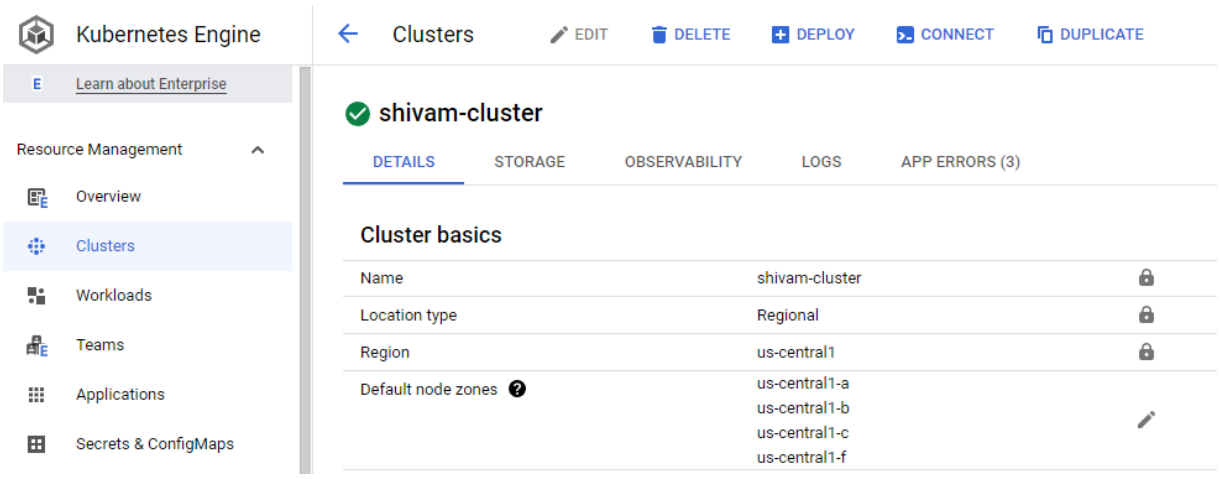
- Different cloud providers have different names for these virtual servers.
  - Amazon calls them EC2 (Elastic Compute Cloud)
  - Azure calls them virtual machines.
  - Google cloud calls them Compute Engines.

Steps to get started with Google Kubernetes Engine

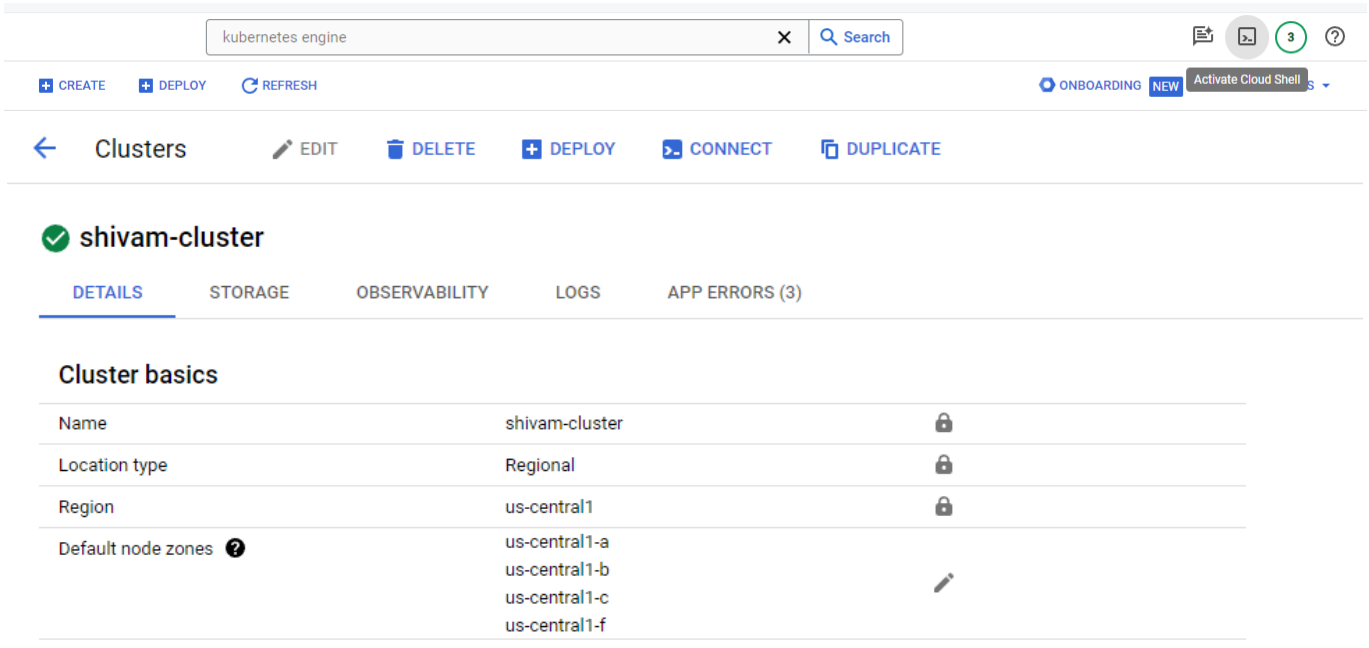
Step 1: Log in to https://cloud.google.com/ & click on Get started for free. Add your details & card info. We will get 300$ credit to use Google cloud service.

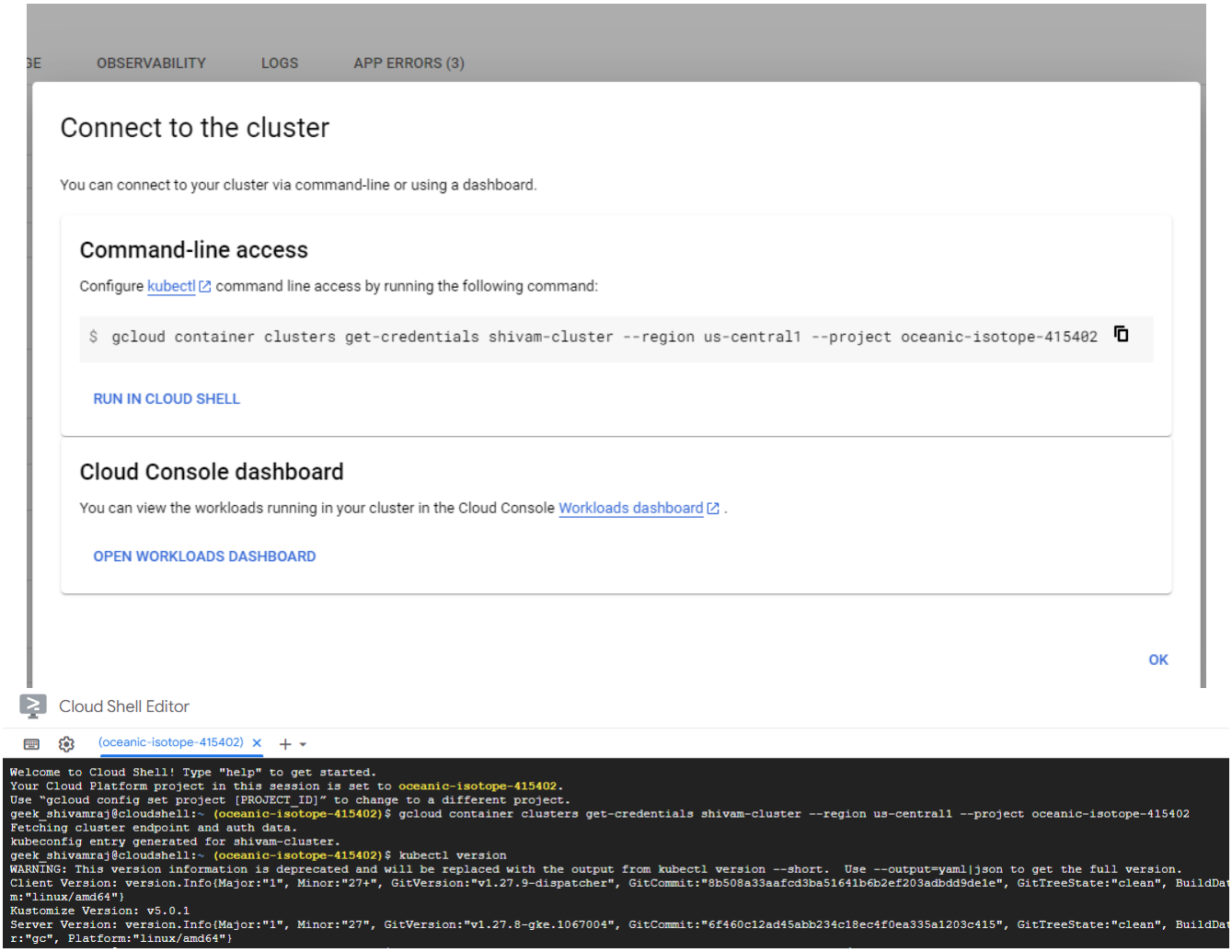
Step 2: Search Kubernetes Engine & create cluster. It will take some time to create cluster.





Step 3: To execute Kubernetes command & explore Kubernetes features, we will **activate cloud shell** & in order to connect with cluster, select the cluster & click on cluster. Copy the command & execute it in cloud shell.





Step 4: To run Kubernetes commands, KUBECTL comes into picture. **kubectl** means **kube controller.**

- **kubectl** is an awesome Kubernetes command to interact with the cluster. It would work with any Kubernetes cluster irrespective of whether the cluster is in our local m/c or in data center or cloud.
- **kubectl** is already installed in the cloud shell.
- We can use **kubectl** command to deploy a new application, increase the no. of instances of an application, deploy newer version of the application.

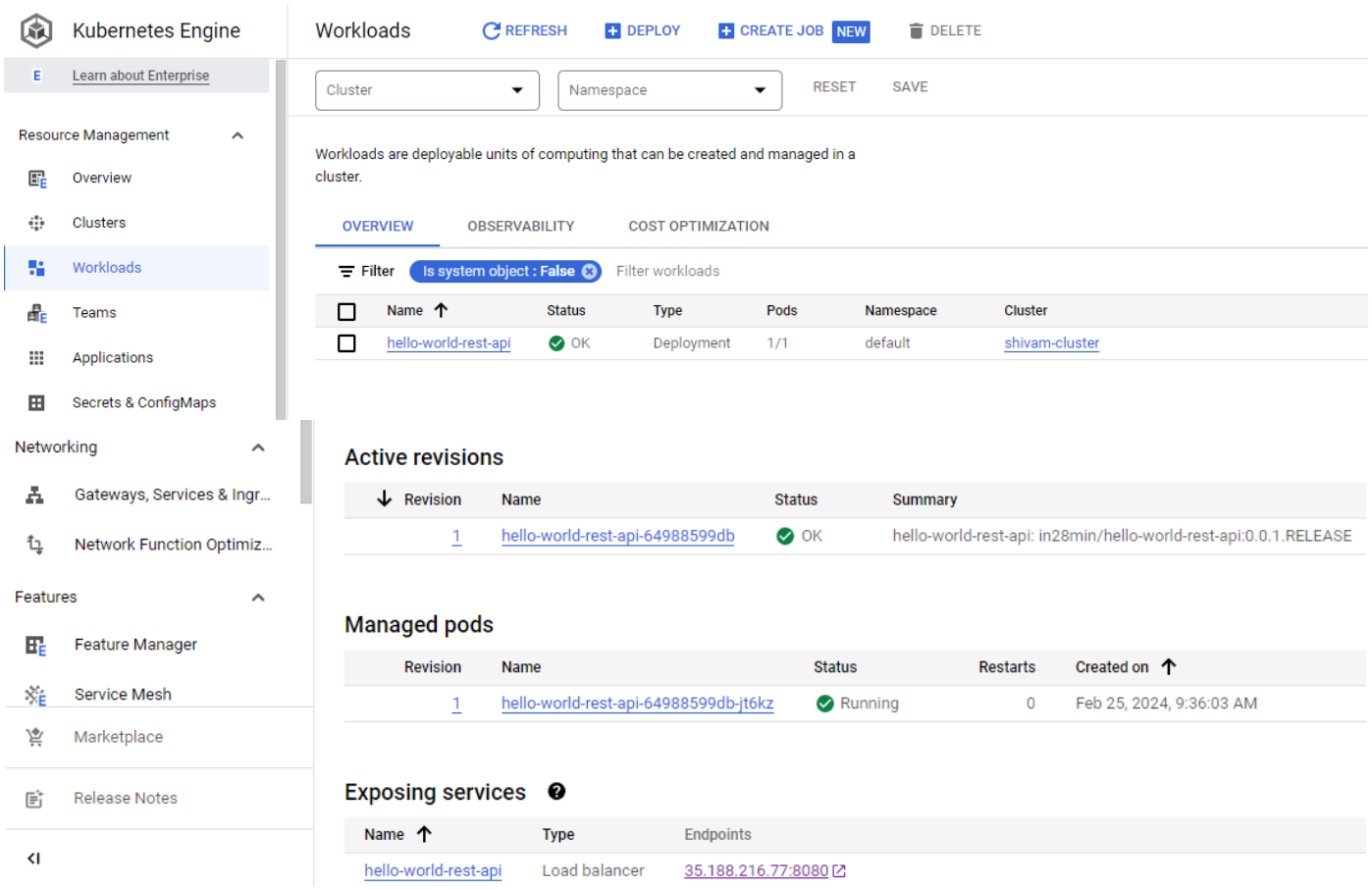
Step 5: To deploy an application, we need to use below commands

- **kubectl create deployment** [deploymentName] --**image** = [imageName:release]
- Docker image will be pulled from docker hub itself.
- e.g., **kubectl create deployment** hello-world-rest-api --**image**=in28min/hello-world-rest-api:0.0.1.RELEASE

Step 6: To expose this deployment to the outside world. We need to execute below commands

- **kubectl expose deployment** [deploymentName] –type=… --port=…
- e.g., **kubectl expose deployment** hello-world-rest-api --**type**=LoadBalancer --**port**=8080

Step 7: Under workloads, we can check our deployment & there we can find the endpoints for exposed services



We can execute URLs for rest services using 35.188.216.77:8080/path
http://35.188.216.77:8080/hello-world
http://35.188.216.77:8080/hello-world-bean

kubectl commands (We can use either singular or plural to get all resources like kubectl get pod / pods)

- **Command** to get events like cluster creation, application startup, pod creation, image pulling, container creation etc.
  - kubectl get events

Imp. point:

- Command to get all created pods, replica set, deployment & service
  - kubectl get pods
  - kubectl get replicaset
  - kubectl get deployment
  - kubectl get service

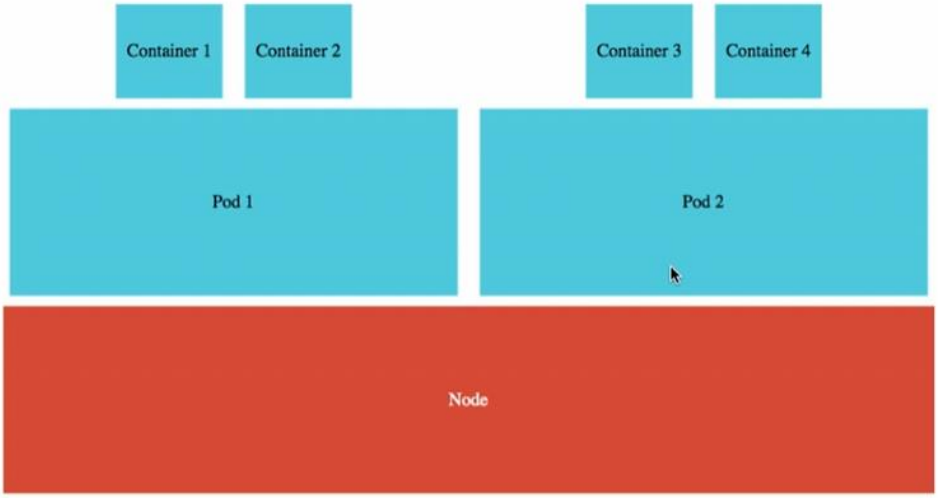
Kubernetes Concept (4 imp. concepts are pod, replicaset, deployment, service)

- The important thing to understand here is Kubernetes uses **Single Responsibility Principle** i.e., One concept One responsibility i.e., each of these 4 concepts have one important responsibility that they do well.
- The combination of these 4 concepts provides the features of Kubernetes i.e., to provide external access to workloads, to enable scaling & to enable zero downtime deployments.

kubectl create deployment .... -> deployment, replicaset & pod
kubectl expose deployment ... -> service

1. Understanding Pods

- A pod is the smallest deployable unit in Kubernetes.
- In Docker, container is the smallest deployable unit but In Kubernetes, Pods are the smallest.
- We can't have a container in Kubernetes without a pod. Our container lives inside a pod.



- Pod is a collection of containers that can run on a host. This resource is created by clients and scheduled onto hosts.
- These pods can be from different applications, or related to the same applications.
- Commands:
  - Kubectl explain pod
  - kubectl get pods / kubectl get pods -o wide
  - kubectl describe pod [podName]

- Each pod has own namespace like default or something. **Namespace** is a very important concept; it provides isolations for parts of the cluster from other parts of the cluster.
- Let's say we have DEV & QA environment running in the same cluster then how do we separate the resources of DEV & QA. One of the options is to create separate namespaces for QA & DEV and associate each resource with that specific namespace.
- In Kubernetes, we link all of the 4 concepts by using something called **Selectors** & **Labels**. Labels are used in tying up a pod with a replicaset or a service.
- **Annotations** are typically meta information about the specific pod like release id, build id, author name etc.
- Each pod has status & IP address. Using IP address, pod provides a way to put our containers together & also it provides a categorization for all these containers by associating them with labels.

```
geek_shivamraj@cloudshell:~$ kubectl get pods -o wide
NAME                                READY   STATUS    RESTARTS   AGE   IP          NODE
hello-world-rest-api-64988599db-nn5lb   1/1     Running   0          13h   10.28.0.6   gk3-demo-cluster-pool-2-687b2efc-npnf
geek_shivamraj@cloudshell:~$ kubectl describe pod hello-world-rest-api-64988599db-nn5lb
Name:             hello-world-rest-api-64988599db-nn5lb
Namespace:        default
Priority:         0
Service Account:  default
Node:             gk3-demo-cluster-pool-2-687b2efc-npnf/10.128.0.7
Start Time:       Sun, 25 Feb 2024 12:40:42 +0000
Labels:           app=hello-world-rest-api
                  pod-template-hash=64988599db
Annotations:      <none>
Status:           Running
SeccompProfile:   RuntimeDefault
IP:               10.28.0.6
IPs:
  IP:           10.28.0.6
Controlled By:  ReplicaSet/hello-world-rest-api-64988599db
Containers:
  hello-world-rest-api:
    Container ID:   containerd://3d83a4eeba3205bc92ebac8ec0e4788f846eec0b8a6718dfa99be1f6d6f39b4f
    Image:          in28min/hello-world-rest-api:0.0.1.RELEASE
    Image ID:       docker.io/in28min/hello-world-rest-api@sha256:00469c343814aabe56ad1034427f546d43bafaaa11208a1eb072099374
```

2. Understanding Replica set

- We can use these commands to fetch replica sets
  - kubectl explain replicaset / replicasets / rs
  - kubectl get replicasets / replicaset / rs
  - kubectl get pods -o wide
  - kubectl delete pods [podName]
  - kubectl get deployment
  - kubectl scale deployment [deploymentName] --replicas=[desiredReplicas]
  - kubectl get events --sort-by=.metadata.creationTimestamp

- **ReplicaSet** ensures that a specified number of pod replicas are running at any given time.
- Even if we kill the running pod, a new pod will be recreated to match the desired no. of pods.
- The replica set always keeps monitoring the pods & if there are lesser no. of pods than what's desired then it creates the pods.
- If desired no. of pods is 1 then if the pod goes down, immediately the replica set looks at it & it will start new pod.

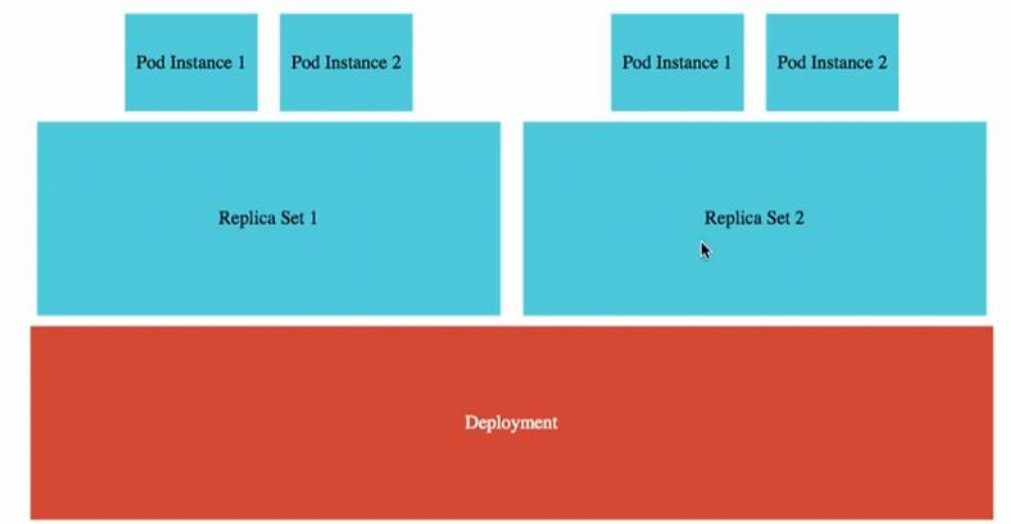
```
geek_shivamraj@cloudshell:~$ kubectl get rs
NAME                              DESIRED   CURRENT   READY   AGE
hello-world-rest-api-64988599db   1         1         1       14h
geek_shivamraj@cloudshell:~$ kubectl get pods -o wide
NAME                                    READY   STATUS    RESTARTS   AGE    IP           NODE
hello-world-rest-api-64988599db-t886f   1/1     Running   0          107s   10.28.0.22   gk3-demo-cluster-pool-2-687b2efc-npnf
geek_shivamraj@cloudshell:~$ kubectl delete pods hello-world-rest-api-64988599db-t886f
pod "hello-world-rest-api-64988599db-t886f" deleted
geek_shivamraj@cloudshell:~$ kubectl get pods -o wide
NAME                                    READY   STATUS    RESTARTS   AGE   IP           NODE
hello-world-rest-api-64988599db-4c7g5   1/1     Running   0          8s    10.28.0.23   gk3-demo-cluster-pool-2-687b2efc-npnf
geek_shivamraj@cloudshell:~$
```

- We can scale up / increase no. of pods. The deployment updates the replica set & says it needs new desired no. of pods.
- Internally, the desired replica set got updated to 3 => then replica set launches other 2 instances/pods as 1 instance is already running.
- We can view all these using command: kubectl get events --sort-by=.metadata.creationTimestamp

```
geek_shivamraj@cloudshell:~$ kubectl get pods -o wide
NAME                                    READY   STATUS    RESTARTS   AGE     IP           NODE
hello-world-rest-api-64988599db-4c7g5   1/1     Running   0          9m32s   10.28.0.23   gk3-demo-cluster-pool-2-687b2efc-npnf
geek_shivamraj@cloudshell:~$ kubectl get rs
NAME                              DESIRED   CURRENT   READY   AGE
hello-world-rest-api-64988599db   1         1         1       14h
geek_shivamraj@cloudshell:~$ kubectl get deployment
NAME                 READY   UP-TO-DATE   AVAILABLE   AGE
hello-world-rest-api   1/1     1            1           14h
geek_shivamraj@cloudshell:~$ kubectl scale deployment hello-world-rest-api --replicas=3
deployment.apps/hello-world-rest-api scaled
geek_shivamraj@cloudshell:~$ kubectl get pods -o wide
NAME                                    READY   STATUS    RESTARTS   AGE   IP           NODE
hello-world-rest-api-64988599db-4c7g5   1/1     Running   0          11m   10.28.0.23   gk3-demo-cluster-pool-2-687b2efc-npnf
hello-world-rest-api-64988599db-6xsb9   0/1     Pending   0          12s   <none>       <none>
hello-world-rest-api-64988599db-wwhht   0/1     Pending   0          12s   <none>       <none>
geek_shivamraj@cloudshell:~$ kubectl get pods -o wide
NAME                                    READY   STATUS    RESTARTS   AGE    IP           NODE
hello-world-rest-api-64988599db-4c7g5   1/1     Running   0          14m    10.28.0.23   gk3-demo-cluster-pool-2-687b2efc-npnf
hello-world-rest-api-64988599db-d516t   1/1     Running   0          2m     10.28.0.67   gk3-demo-cluster-pool-2-047e804a-4ql8
hello-world-rest-api-64988599db-wwhht   1/1     Running   0          3m9s   10.28.0.66   gk3-demo-cluster-pool-2-047e804a-4ql8
geek_shivamraj@cloudshell:~$ kubectl get rs
NAME                              DESIRED   CURRENT   READY   AGE
hello-world-rest-api-64988599db   3         3         3       14h
geek_shivamraj@cloudshell:~$ kubectl get deployment
NAME                 READY   UP-TO-DATE   AVAILABLE   AGE
hello-world-rest-api   3/3     3            3           14h
geek_shivamraj@cloudshell:~$
```

3. Understand Deployment

- Deployment enables declarative updates for Pods and ReplicaSets.
- Deployment makes sure that we're able to update new release of applications without downtime.
- The deployment default strategy is Rolling updates i.e., it updates one pod at a time.



- Deployment commands:
  - kubectl explain deployment
  - kubectl get deployment
  - kubectl get deployment -o wide
  - kubectl describe pod [podName]
  - kubectl set image deployment [deploymentName] [containerName]=imageName:release

- Let's say we want to update the application to a new version i.e., from V1 to V2 & we also want zero downtime.
- Here V1 image is in28min/hello-world-rest-api:0.0.1.RELEASE & the replica set is managing this specific image release.

```
geek_shivamraj@cloudshell:~$ kubectl get pods -o wide
NAME                                  READY   STATUS    RESTARTS   AGE   IP           NODE                                          NOMINA
TES
hello-world-rest-api-64988599db-4c7g5   1/1     Running   0          72m   10.28.0.23   gk3-demo-cluster-pool-2-687b2efc-npnf   <none>
hello-world-rest-api-64988599db-d516t   1/1     Running   0          59m   10.28.0.67   gk3-demo-cluster-pool-2-047e804a-4ql8   <none>
hello-world-rest-api-64988599db-wwhht   1/1     Running   0          60m   10.28.0.66   gk3-demo-cluster-pool-2-047e804a-4ql8   <none>
geek_shivamraj@cloudshell:~$ kubectl get deployment -o wide
NAME                READY   UP-TO-DATE   AVAILABLE   AGE   CONTAINERS             IMAGES                                        SELE
hello-world-rest-api   3/3     3            3           15h   hello-world-rest-api   in28min/hello-world-rest-api:0.0.1.RELEASE   app=
geek_shivamraj@cloudshell:~$ kubectl get rs -o wide
NAME                                DESIRED   CURRENT   READY   AGE   CONTAINERS             IMAGES
hello-world-rest-api-64988599db   3         3         3       15h   hello-world-rest-api   in28min/hello-world-rest-api:0.0.1.RELEASE
```

- We can set a new release as V2 using below command & also even if we give wrong name or release, already running pods with V1 release won't go down.
  - **kubectl set image deployment** hello-world-rest-api hello-world-rest-api=DUMMY_IMAGE:TEST

```
ranga@cloudshell:~ (solid-course-258105)$ kubectl get rs -o wide
NAME                                DESIRED   CURRENT   READY   AGE   CONTAINERS             IMAGES
                                  SELECTOR
hello-world-rest-api-58ff5dd898   3         3         3       49m   hello-world-rest-api   in28min/hello-
world-rest-api:0.0.1.RELEASE   app=hello-world-rest-api,pod-template-hash=58ff5dd898
hello-world-rest-api-85995ddd5c   1         1         0       62s   hello-world-rest-api   DUMMY_IMAGE:TE
ST                             app=hello-world-rest-api,pod-template-hash=85995ddd5c
ranga@cloudshell:~ (solid-course-258105)$ kubectl get pods
NAME                                READY   STATUS             RESTARTS   AGE
hello-world-rest-api-58ff5dd898-5x4vl   1/1     Running            0          14m
hello-world-rest-api-58ff5dd898-vn7d6   1/1     Running            0          16m
hello-world-rest-api-58ff5dd898-zp2nq   1/1     Running            0          14m
hello-world-rest-api-85995ddd5c-msjsm   0/1     InvalidImageName   0          2m6s
```

- We can see a new replicaset will be created with V2 version & now the new replicaset will try to launch one instance of the pod (this pod is in failed status).

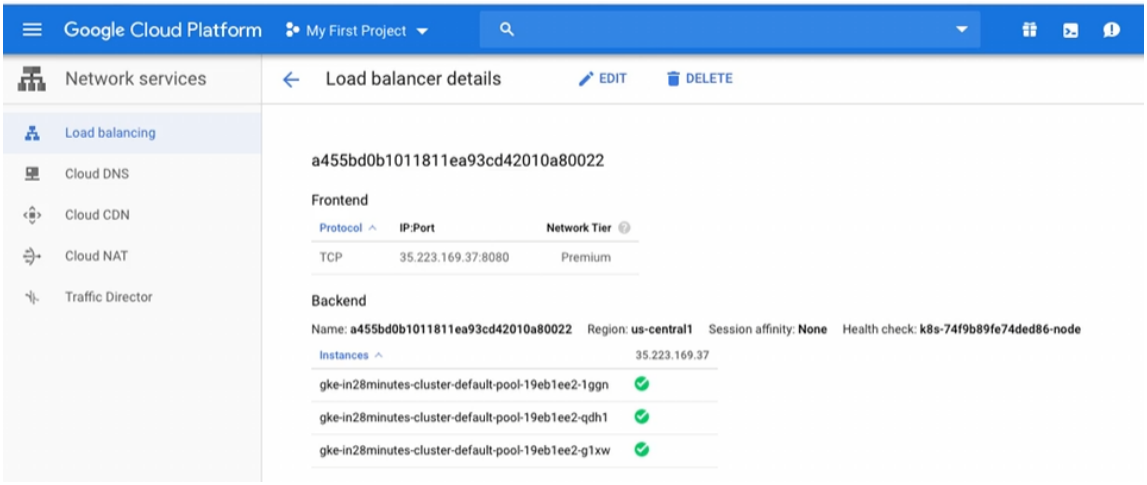
- Now let's run a valid case
  - kubectl set image deployment hello-world-rest-api hello-world-rest-api=in28min/hello-world-rest-api:0.0.2.RELEASE

- In background,
  Step1: Deployment creates a new version of replica set for V2 & ask replica set to create a pod.
  Step2: The replica set immediately try to create a pod.
  Step3: Once the pod is created successfully. Deployment will say okay I have one pod of V2 up & running. I don't need 3 instances anymore. It scales down the V1 replica from 3 to 2 & scales up V2 replica from 1 to 2 & same process goes on until we have 3 V2 replicas & 0 V1 replicas.
  Step4: This deployment default strategy is called **Rolling updates.**

4. Understanding Services

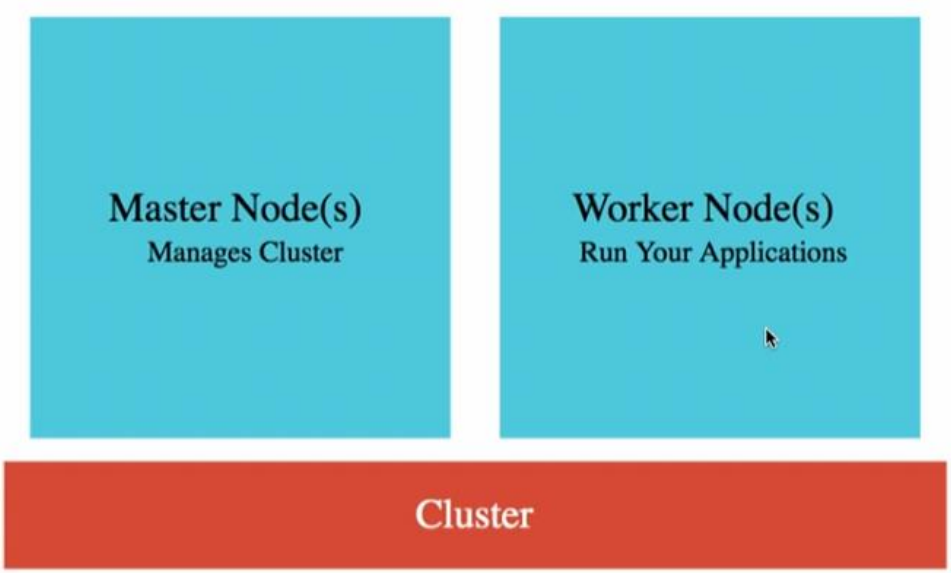
- Service is a named abstraction of software service (for example, mysql) consisting of local port (for example 3306) that the proxy listens on, and the selector that determines which pods will answer requests sent through the proxy.
- IP address is specific to pod. So, every time a new pod will be created, a new IP address will be initialized to it.
- Now, let's think from Consumer's perspective, we would want a single URL of the application irrespective of the pods in the background. That's where service comes.
- The role of a service is to provide an always available external interface to the applications which are running inside the pods.
- A service basically allows our application to receive traffic through a permanent lifetime IP address.
- The service is created with an external IP address when we expose deployment.
  - **kubectl expose deployment** hello-world-rest-api --type=LoadBalancer --port=8080
  - **kubectl get service -o wide**

- Here we're creating a Google cloud Load balancer service. All the changes come to backend but front end still remain same. A load balancer can load balance b/w multiple pods.



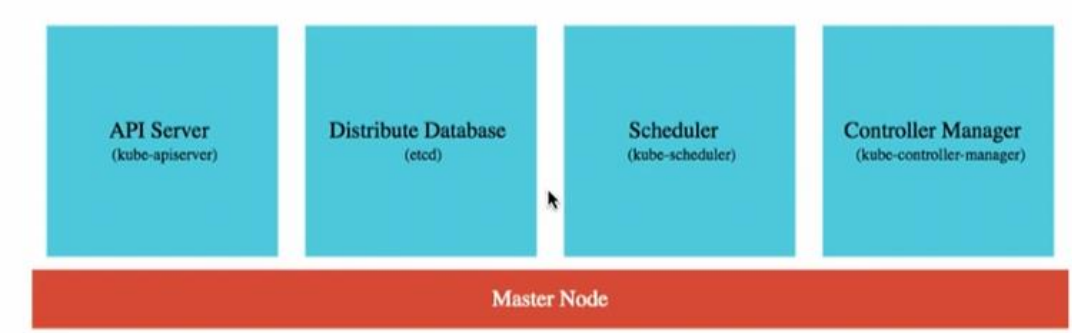
- Kubernetes service is actually running as a cluster IP service. A cluster IP service can only be accessed from inside the cluster. We won't be able to access this service from outside the cluster. We can see no external-ip to kubernetes service.

Kubernetes Architecture (Master Nodes & Worker Nodes/Nodes)



- The important thing about master node is all the user applications like hello-world-rest-api etc. would be running in PODS inside the worker nodes or just nodes.

Master Node



1. Distributed Database (ETCD)

- The most important component of master node is Distributed Database (etcd)
- All the deployment resources like deployment services, configuration changes, deployment creation, scaling operation, desired state etc. all these details are stored in a distributed database.
- This database is distributed. Typically it's recommended to have 3 to 5 replicas of this database so that the Kubernetes cluster state is not lost.

2. API Server (kube – apiserver)

- Kubectl commands, Google cloud interface or Google cloud console all talk or make their changes to Kubernetes cluster through API server.

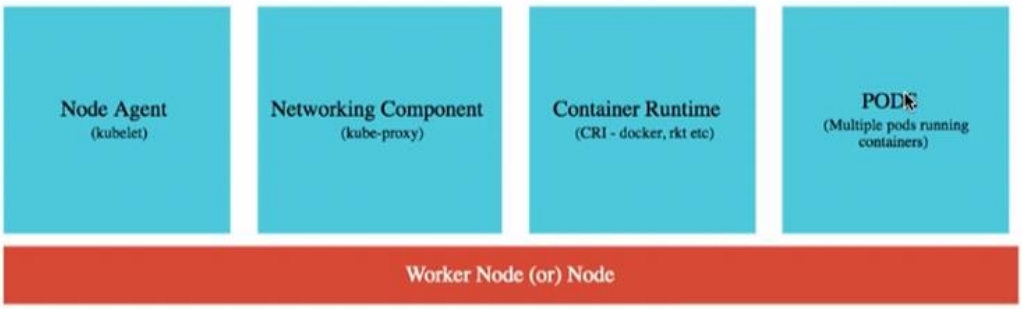
3. Scheduler (kube – scheduler)

- The Scheduler is responsible for scheduling the pods onto the nodes.
- i.e., In a Kubernetes cluster, we'll have several nodes & when we're creating a new pod, we need to decide which node the pod has to be scheduled on to.
- The decision might be based on how much memories, CPU available or is there any pod conflicts & lot of factors. So, Scheduler considers all the factors & schedules the pods onto the appropriate Nodes.

4. Controller Manager (kube – controller – manager)

- The Controller manager manages the overall health of the cluster. It makes sure that the actual state of cluster matches the desired state.

Worker Node or Node



1. PODS

- All the user application will be running inside the PODS on a single node.

2. Node Agent (kubelet)

- The Node agent monitors what's happening on the Node & communicates it back to Master node.
- For e.g., if a pod goes down, node agent will report it to the Controller manager.

3. Network Component (kube – proxy)

- Because of the network component, we can expose the deployment as a service.
- i.e., it helps us in exposing services around out nodes & pods.

4. Container Runtime (CRI – docker, rkt etc)

- We need container runtime to run containers inside our PODS.
- We can use Kubernetes with any OCI (Open Container Interface) runtime spec implementations. Docker is one of them.

Important Conceptual Questions

Q1. Does a Master Node run any of the application related containers?
Ans: No, Master node is typically having only the stuff to control our Worker nodes.

Q2. Can we only run Docker containers in Kubernetes?
Ans: No, we can run any container that is compatible with OCI (Open Container interface) in Kubernetes.

Q3. What happens if the Master node goes down or specific service on master node goes down? Will the application go down?
Ans: No, the application can continue to run working even with the Master node down. When we're executing the URL to access an application, the Master node doesn't get involved at all only Worker nodes will be involved. We can't make changes to them but the existing application would continue to run.

```
ranga@cloudshell:~ (solid-course-258105)$ kubectl get componentstatuses
NAME                 STATUS    MESSAGE             ERROR
etcd-0               Healthy   {"health": "true"}
controller-manager   Healthy   ok
scheduler            Healthy   ok
etcd-1               Healthy   {"health": "true"}
ranga@cloudshell:~ (solid-course-258105)$
```

Deploying Microservices using Kubernetes in Google Kubernetes Engine

- We need to follow just below step in order to create & expose our microservices

Step1: Build the image for the microservices & push it to the docker hub. Now run: docker images

```
root@LAPTOP-J489HHHD:/home/shivam# docker images
REPOSITORY                                                TAG              IMAGE ID
geekshivamraj/k8s-microservices-currency-conversion-service   0.0.11-SNAPSHOT   5904494c0658
geekshivamraj/k8s-microservices-currency-exchange-service     0.0.11-SNAPSHOT   d8d2017d2ebe
root@LAPTOP-J489HHHD:/home/shivam#
```

Step2: Now using kubectl command we will check the version: kubectl version

```
root@LAPTOP-J489HHHD:/home/shivam# kubectl version
WARNING: This version information is deprecated and will be replaced with the output from kubectl version --short.  Use --output=yaml|j
son to get the full version.
Client Version: version.Info{Major:"1", Minor:"27", GitVersion:"v1.27.2", GitCommit:"7f6f68fdabc4df88cfea2dcf9a19b2b830f1e647", GitTree
State:"clean", BuildDate:"2023-05-17T14:20:07Z", GoVersion:"go1.20.4", Compiler:"gc", Platform:"linux/amd64"}
Kustomize Version: v5.0.1
Server Version: version.Info{Major:"1", Minor:"27", GitVersion:"v1.27.8-gke.1067004", GitCommit:"6f460c12ad45abb234c18ec4f0ea335a1203c4
15", GitTreeState:"clean", BuildDate:"2024-01-04T22:48:32Z", GoVersion:"go1.20.11 X:boringcrypto", Compiler:"gc", Platform:"linux/amd64
"}
root@LAPTOP-J489HHHD:/home/shivam#
```

Note: If you're not able to see server version i.e., you are yet to login to google cloud using terminal. Go to the cluster & click on connect, you will get a command, just copy & paste here. Now check again: kubectl version

Step3: Now we will be creating deployment for each microservices

a) currency-exchange microservice

- **kubectl create deployment** currency-exchange --image=geekshivamraj/k8s-microservices-currency-exchange-service:0.0.11-SNAPSHOT
- When you run above command: Deployment created => Replicaset created => Pod created

- But service is yet not exposed to outside world. To expose service run below command:
- **kubectl expose deployment** currency-exchange --type=LoadBalancer --port=8000
- When we run above: Deployment created => Replicaset created => Pod created => Service exposed
- **Url**: http://34.16.9.92:8000/currency-exchange/from/USD/to/INR

b) currency-conversion microservice

- **kubectl create deployment** currency-conversion --image=geekshivamraj/k8s-microservices-currency-conversion-service:0.0.11-SNAPSHOT
- **kubectl expose** deployment currency-conversion --type=LoadBalancer --port=8100
- **Url:** http://35.232.135.76:8100/currency-conversion-feign/from/USD/to/INR/quantity/10

```
root@LAPTOP-J489HHHD:/home/shivam# kubectl get deployment
NAME                  READY   UP-TO-DATE   AVAILABLE   AGE
currency-conversion   1/1     1            1           27m
currency-exchange     1/1     1            1           40m
```

```
root@LAPTOP-J489HHHD:/home/shivam# kubectl get svc
NAME                  TYPE           CLUSTER-IP       EXTERNAL-IP     PORT(S)          AGE
currency-conversion   LoadBalancer   34.118.226.176   35.232.135.76   8100:30426/TCP   27m
currency-exchange     LoadBalancer   34.118.239.124   34.16.9.92      8000:31077/TCP   35m
kubernetes            ClusterIP      34.118.224.1     <none>          443/TCP          78m
```

```
root@LAPTOP-J489HHHD:/home/shivam# kubectl get pods
NAME                                  READY   STATUS    RESTARTS   AGE
currency-conversion-79cbc5769d-cslnw  1/1     Running   0          30m
currency-exchange-686cf7d659-jrbzt    1/1     Running   0          39m
```

Important point to note:

- When a service is created in kubernetes, along with that environment variables are also created like
  - Suppose we've a service with name: service-name then Kubernetes will create an environment variable with the name as: **SERVICE_NAME_SERVICE_HOST**
- Whenever we launch up a new port, all the existing services information is made available to the port as environment variables.
- We're using some of the services environment variables in our code:
  - **CURRENCY_EXCHANGE_SERVICE_HOST**
  - **HOSTNAME**

```
@FeignClient(name = "currency-exchange", url = "${CURRENCY_EXCHANGE_SERVICE_HOST:http://localhost}:8000")
//@FeignClient(name = "currency-exchange", url = "${CURRENCY_EXCHANGE_URI:http://localhost}:8000")
public interface CurrencyExchangeProxy {
    1 usage
    @GetMapping("/currency-exchange/from/{from}/to/{to}")
    public CurrencyConversion retrieveExchangeValue(@PathVariable String from, @PathVariable String to);
}
```

```
// CHANGE-KUBERNETES
String host = environment.getProperty("HOSTNAME");
String version = "v11";
currencyExchange.setEnvironment(port + " " + version + " " + host);
return currencyExchange;
```

- **kubectl exec** -it [podName] -- env

```
root@LAPTOP-J489HHHD:/home/shivam# kubectl exec -it currency-conversion-79cbc5769d-cslnw -- env
PATH=/cnb/process:/cnb/lifecycle:/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin
HOSTNAME=currency-conversion-79cbc5769d-cslnw
CNB_LAYERS_DIR=/layers
CNB_APP_DIR=/workspace
CNB_PLATFORM_API=0.12
CNB_DEPRECATION_MODE=quiet
CURRENCY_EXCHANGE_PORT=tcp://34.118.239.124:8000
KUBERNETES_SERVICE_PORT_HTTPS=443
KUBERNETES_PORT_443_TCP_PROTO=tcp
KUBERNETES_PORT_443_TCP_PORT=443
KUBERNETES_PORT_443_TCP_ADDR=34.118.224.1
CURRENCY_EXCHANGE_SERVICE_HOST=34.118.239.124
KUBERNETES_SERVICE_HOST=34.118.224.1
KUBERNETES_PORT=tcp://34.118.224.1:443
CURRENCY_EXCHANGE_SERVICE_PORT=8000
CURRENCY_EXCHANGE_PORT_8000_TCP_PROTO=tcp
KUBERNETES_PORT_443_TCP=tcp://34.118.224.1:443
CURRENCY_EXCHANGE_PORT_8000_TCP=tcp://34.118.239.124:8000
CURRENCY_EXCHANGE_PORT_8000_TCP_ADDR=34.118.239.124
KUBERNETES_SERVICE_PORT=443
CURRENCY_EXCHANGE_PORT_8000_TCP_PORT=8000
TERM=xterm
HOME=/home/cnb
root@LAPTOP-J489HHHD:/home/shivam#
```

```
root@LAPTOP-J489HHHD:/home/shivam# kubectl exec -it currency-exchange-686cf7d659-jrbzt -- env
PATH=/cnb/process:/cnb/lifecycle:/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin
HOSTNAME=currency-exchange-686cf7d659-jrbzt
CNB_LAYERS_DIR=/layers
CNB_APP_DIR=/workspace
CNB_PLATFORM_API=0.12
CNB_DEPRECATION_MODE=quiet
KUBERNETES_PORT_443_TCP_PROTO=tcp
KUBERNETES_PORT_443_TCP_PORT=443
KUBERNETES_PORT_443_TCP_ADDR=34.118.224.1
KUBERNETES_SERVICE_HOST=34.118.224.1
KUBERNETES_SERVICE_PORT=443
KUBERNETES_SERVICE_PORT_HTTPS=443
KUBERNETES_PORT=tcp://34.118.224.1:443
KUBERNETES_PORT_443_TCP=tcp://34.118.224.1:443
TERM=xterm
HOME=/home/cnb
root@LAPTOP-J489HHHD:/home/shivam#
```

Declarative Configuration Kubernetes YAML for microservices

- So far, we've used commands to deploy our microservices in Kubernetes. However, Kubernetes also provides a declarative configuration of deploying applications by using YAML file.
-

Step1: Get deployment.yaml & service.yaml from Kubernetes environment using below commands

- kubectl get deployment currency-exchange -o yaml >> deployment.yaml
- kubectl get service currency-exchange -o yaml >> service.yaml

Step2: Copy the content of service.yaml & append it to deployment.yaml & delete service.yaml.
Step3: For e.g., we can increase replica for currency-exchange microservice.

To check deployment.yaml diff (once we make any change) & apply deployment.yaml

- kubectl diff -f deployment.yaml
- kubectl apply -f deployment.yaml

```
10      labels:
11        app: currency-exchange
12      name: currency-exchange
13      namespace: default
14      resourceVersion: "30383"
15      uid: d54b2412-7115-4030-9a1f-3e59f87b6d05
16    spec:
17      progressDeadlineSeconds: 600
18      replicas: 2
19      revisionHistoryLimit: 10
20      selector:
21        matchLabels:
22          app: currency-exchange
23      strategy:
```

```
root@LAPTOP-J489HHHD:/mnt/d/IntellijWs/MicroServicesCourse/k8s-currencyMicroservice/currency-exchange-service# kubectl get pods
NAME                                  READY   STATUS    RESTARTS   AGE
currency-conversion-79cbc5769d-krf49  1/1     Running   0          76s
currency-exchange-686cf7d659-jrbzt    1/1     Running   0          164m
currency-exchange-686cf7d659-ssnf5    1/1     Running   0          2m8s
```

- **Url**: http://34.16.9.92:8000/currency-exchange/from/USD/to/INR

```
root@LAPTOP-J489HHHD:/home/shivam# curl http://34.16.9.92:8000/currency-exchange/from/USD/to/INR
{"id":10001,"from":"USD","to":"INR","conversionMultiple":65.00,"environment":"8000 v11 currency-exchange-686cf7d659-jrbzt"}
root@LAPTOP-J489HHHD:/home/shivam# curl http://34.16.9.92:8000/currency-exchange/from/USD/to/INR
{"id":10001,"from":"USD","to":"INR","conversionMultiple":65.00,"environment":"8000 v11 currency-exchange-686cf7d659-ssnf5"}
```

- Above, we can see environment with different pod names.
- So, in addition to providing service discovery environment variables, Kubernetes also provides load balancing between these services.
- So, very important 2 features of microservices are provided for us for free.

Clean up Kubernetes YAML for Microservices

- So, in addition to providing service discovery environment variables, Kubernetes also provides load balancing between these services.
- Let's delete all the existing deployment & pods
  - kubectl delete all -l app=currency-exchange
  - kubectl delete all -l app=currency-conversion

```
root@LAPTOP-J489HHHD:/home/shivam# kubectl get all -o wide
NAME                 TYPE        CLUSTER-IP     EXTERNAL-IP   PORT(S)   AGE     SELECTOR
service/kubernetes   ClusterIP   34.118.224.1   <none>        443/TCP   5h33m   <none>
root@LAPTOP-J489HHHD:/home/shivam#
```

deployment.yaml for currency-exchange microserive

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  annotations:
    deployment.kubernetes.io/revision: "2"
  labels:
    app: currency-exchange
  name: currency-exchange
  namespace: default
spec:
  replicas: 1
  selector:
    matchLabels:
      app: currency-exchange
  strategy:
    rollingUpdate:
      maxSurge: 25%
      maxUnavailable: 25%
    type: RollingUpdate
  template:
    metadata:
      labels:
        app: currency-exchange
    spec:
      containers:
        - image: geekshivamraj/k8s-microservices-currency-exchange-service:0.0.11-SNAPSHOT
          imagePullPolicy: Always
          name: k8s-microservices-currency-exchange-service
      restartPolicy: Always

---

apiVersion: v1
kind: Service
metadata:
  labels:
    app: currency-exchange
  name: currency-exchange
  namespace: default
spec:
  ports:
    - port: 8000
      protocol: TCP
      targetPort: 8000
  selector:
    app: currency-exchange
  sessionAffinity: None
  type: LoadBalancer
```

deployment.yaml for currency-conversion microservice

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  annotations:
    deployment.kubernetes.io/revision: "1"
  labels:
    app: currency-conversion
  name: currency-conversion
  namespace: default
spec:
  replicas: 1
  selector:
    matchLabels:
      app: currency-conversion
  strategy:
    rollingUpdate:
      maxSurge: 25%
      maxUnavailable: 25%
    type: RollingUpdate
  template:
    metadata:
      labels:
        app: currency-conversion
    spec:
      containers:
        - image: geekshivamraj/k8s-microservices-currency-conversion-service:0.0.11-SNAPSHOT
          imagePullPolicy: Always
          name: k8s-microservices-currency-conversion-service
      restartPolicy: Always

---

apiVersion: v1
kind: Service
metadata:
  labels:
    app: currency-conversion
  name: currency-conversion
  namespace: default
spec:
  ports:
    - port: 8100
      protocol: TCP
      targetPort: 8100
  selector:
    app: currency-conversion
  sessionAffinity: None
  type: LoadBalancer
```

Note: These are the minimum configurations we need to give in order for Kubernetes to create deployment & expose services. To check deployment.yaml diff (once we make any change) & apply deployment.yaml (run below commands from the same directory container deployment.yaml)

- kubectl diff -f deployment.yaml
- kubectl apply -f deployment.yaml

Issue with previous approach of using pod environment variable created by Kubernetes:

- The problem with the way we have implemented the **service discovery** & load balancing. The approach we took in our currency-exchange proxy was to use currency-exchange service environment variable **(CURRENT_EXCHANGE_SERVICE_HOST**) that is dependent on the pod creation.
- There can be the case where we're starting up currency-conversion microservice but at that particular point of time, currency-exchange microservice is not available. In this case, the currency-conversion service will not get the currency-exchange service host environment variable.

CurrencyExchangeProxy.java

```
package com.srvcode.microservices;

import ...

1 usage
@FeignClient(name = "currency-exchange", url = "${CURRENCY_EXCHANGE_SERVICE_HOST:http://localhost}:8000")
public interface CurrencyExchangeProxy {
    1 usage
    @GetMapping("/currency-exchange/from/{from}/to/{to}")
    public CurrencyConversion retrieveExchangeValue(@PathVariable String from, @PathVariable String to);
}
```

- That's the reason why it's recommended to create custom environment variables instead of using the default environment variables provided by Kubernetes.

- Let's create our custom environment variable with name "CURRENCY_EXCHANGE_URI" & provide its reference in deployment.yaml

  Note: Whenever we're making any change in code, we need to update snapshot for new docker image.

CurrencyExchangeProxy.java

```
package com.srvcode.microservices;

import ...

1 usage
@FeignClient(name = "currency-exchange", url = "${CURRENCY_EXCHANGE_URI:http://localhost}:8000")
public interface CurrencyExchangeProxy {
    1 usage
    @GetMapping("/currency-exchange/from/{from}/to/{to}")
    public CurrencyConversion retrieveExchangeValue(@PathVariable String from, @PathVariable String to);
}
```

- We need to update image & add environment variable (CURRENCY_EXCHANGE_URI=http://serviceName)

```
spec:
  containers:
    - image: geekshivamraj/k8s-microservices-currency-conversion-service:0.0.12-SNAPSHOT
      imagePullPolicy: Always
      name: k8s-microservices-currency-conversion-service
      env:
        - name: CURRENCY_EXCHANGE_URI
          value: http://currency-exchange
  restartPolicy: Always
```

- So, Inside kubernetes, whenever we look for http://currency-exchange, we'll be load balanced b/w the existing instances of currency-exchange service.
- Now, we can check diff & deploy the new snapshot.
- **URL:** curl http://35.222.39.72:8100/currency-conversion-feign/from/USD/to/INR/quantity/10

Centralized Configuration in Kubernetes - ConfigMaps

- Previously, we're adding Hardcoded value in deployment.yaml for CURRENCY_EXCHANGE_URI. Instead of that, Kubernetes also provides a centralized configuration option i.e., **ConfigMap**
- Steps to follow for config map creation & adding that to deployment.yaml

Step1: Create Config map & add the data.

- **kubectl create** configmap currency-conversion --from-literal=CURRENCY_EXCHANGE_URI=http://currency-exchange

Step2: We can see config map using below commands

- **kubectl get** configmap
- **kubectl get** configmap currency-conversion
- **kubectl get** configmap currency-conversion -o yaml

```
root@LAPTOP-J489HHHD:/home/shivam# kubectl get configmap currency-conversion -o yaml
apiVersion: v1
data:
  CURRENCY_EXCHANGE_URI: http://currency-exchange
kind: ConfigMap
metadata:
  creationTimestamp: "2024-02-29T12:01:27Z"
  name: currency-conversion
  namespace: default
  resourceVersion: "407570"
  uid: 9b2f131c-fb58-41a4-bc8d-f3c918c69b61
root@LAPTOP-J489HHHD:/home/shivam#
```

Step3: Create the configmap.yaml for currency-conversion

- **kubectl get** configmap currency-conversion -o yaml >> configmap.yaml

Step4: Remove useless info from configmap.yaml & add the remaining config to deployment.yaml

```
52  ---
53  apiVersion: v1
54  data:
55    CURRENCY_EXCHANGE_URI: http://currency-exchange
56  kind: ConfigMap
57  metadata:
58    name: currency-conversion
59    namespace: default
60
```

Step5: Deploy & test

- **URL:** curl http://35.222.39.72:8100/currency-conversion-feign/from/USD/to/INR/quantity/10

Imp. Notes:

- As we saw All the typical features needed by Microservices, are provided for free in Kubernetes like Service discovery, load balancing & Centralized configuration management using ConfigMaps.

Microservice Deployment with Kubernetes

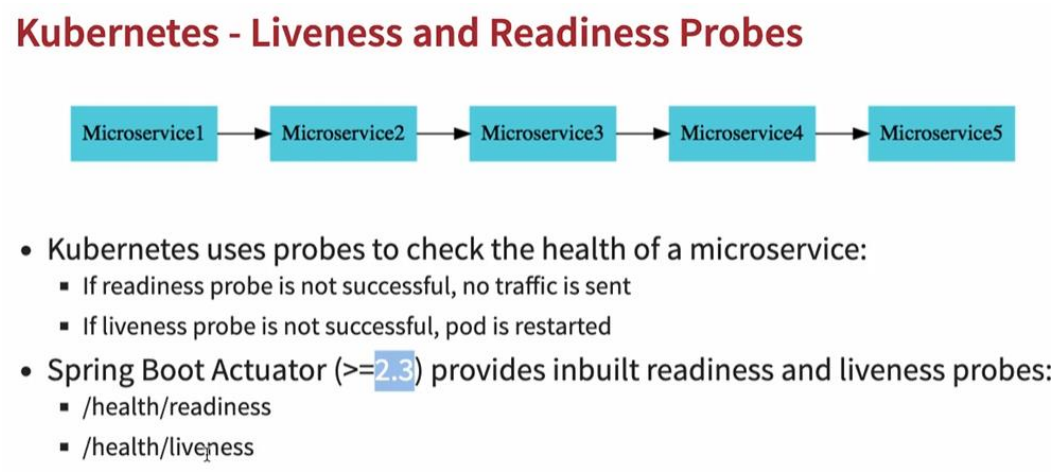
- Kubernetes provides lot of features to ensure that our deployments don't have any downtime.
- We can rollback our deployment to previous revision if required
  - kubectl rollout history deployment currency-conversion
  - kubectl rollout history deployment currency-exchange
  - kubectl rollout undo deployment currency-exchange --to-revision=1

```
rangaraokaranam$ kubectl get pods
NAME                                  READY   STATUS             RESTARTS   AGE
currency-conversion-9b7df7956-hlmlf   1/1     Running            0          23m
currency-exchange-686bbff8dc-sttz6    1/1     Running            0          57m
currency-exchange-dc6c88d9d-2t4f2     0/1     InvalidImageName   0          37s
rangaraokaranam$ kubectl rollout history deployment currency-exchange
deployment.apps/currency-exchange
REVISION  CHANGE-CAUSE
1         <none>
2         <none>

rangaraokaranam$ kubectl rollout undo deployment currency-exchange --to-revision=1
deployment.apps/currency-exchange rolled back
rangaraokaranam$ kubectl get pods
NAME                                  READY   STATUS    RESTARTS   AGE
currency-conversion-9b7df7956-hlmlf   1/1     Running   0          25m
currency-exchange-686bbff8dc-sttz6    1/1     Running   0          58m
```

Imp. Point

- When we're switching from one deployment to the other deployment, we would see that there is a little bit of downtime.
- We can avoid this by using **the Liveness & the Readiness Probes**, provided by Kubernetes. We can configure them to check the health of the microservices.
- These 2 probes are very useful when it comes to making the microservices highly available.
- The amazing thing is Spring Boot Actuator (Spring Boot V2.3 >) provides inbuilt readiness & liveness probes & they are available at /health/readiness & /health/liveness.



- We need to add below configurations to application.properties of microservices

```
10  ## CHANGE-KUBERNETES
11  management.endpoint.health.probes.enabled=true
12  management.health.livenessState.enabled=true
13  management.health.readinessState.enabled=true
```

- We can configure the liveness & readiness probe on the container in deployment.yaml

```yaml
spec:
  containers:
    - image: geekshivamraj/k8s-microservices-currency-exchange-service:0.0.12-SNAPSHOT
      imagePullPolicy: Always
      name: k8s-microservices-currency-exchange-service
      readinessProbe:
        httpGet:
          port: 8000
          path: /actuator/health/readiness
      livenessProbe:
        httpGet:
          port: 8000
          path: /actuator/health/liveness
  restartPolicy: Always
```

Autoscaling Microservices with Kubernetes

- We want to autoscale the microservice based on the load that's coming on it. Initially we were scaling using manual command or manually updating replicas for microservice.
  - kubectl autoscale deployment [deploymentName] parameters
  - kubectl autoscale deployment currency-exchange --min=1 --max=3 --cpu-percent=5
  - kubectl get hpa
  - kubectl top pod
  - kubectl top nodes
  - kubectl delete hpa currency-exchange
  - watch -n 0.1 curl [serviceUrl] – This will keep on making api call

```
root@LAPTOP-J489HHHD:/home/shivam# kubectl get hpa
NAME                REFERENCE                      TARGETS   MINPODS   MAXPODS   REPLICAS   AGE
currency-exchange   Deployment/currency-exchange   7%/5%     1         3         3          7m31s
```

- We can delete the hpa (horizontal pod autoscaler) to stop autoscaling for the particular microservice & then we can apply deployment.yaml to update the configure (replica from 3 to 1)