Kubernetes

Monitoring Tools

Bugzilla - keep track of outstanding bugs, problems, issues, enhancement and other change requests in their products effectively.

JIRA is an open source bug tracking tool.

Docker

Pending Topics

1) Docker save command covert docker image into tar file

Command docker save tomcatimage:latest > tomcat.tar

2) Docker load command create the image from tar file ,we use scp or rsync to copy tar file Into other instance

Command docker load < tomcat.tar

3) We can use images on different instance without docker registery we use **docker save** and **docker load**

4) Command Docker history imagename – shows modification on the image

5) Docker state – use to check CPU and Memory usage of the container

Command docker run -d --memory="600m" --cpus="0.5" -p 8080:8080 tomcatimage:latest

6) Command docker exec = docker attach

7) Command docker commit- create image from running container

8) Docker Security

When using Docker containers, you should use the following practices to ensure maximum security.

1) Avoid Root Permissions
2) Use Secure Container Registries
3) Limit Resource Usage
4) Scan Your Images
5) Build Your Networks and APIs for Security
6) Docker Container Monitoring

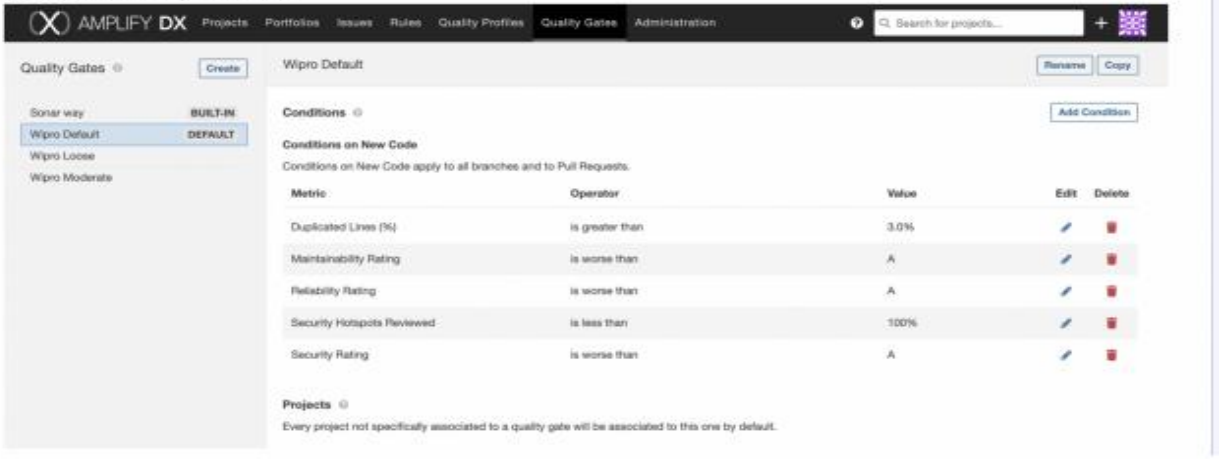
---

1) SonarQube is used as a vulnerability/security issues check and test coverage tool. It has a user-friendly interface for developers, even suggest alternatives for each code smell

2) What are quality gates in sonar-qube ?

Quality Gates are the set of conditions a project must meet before it should be pushed to further environments. Quality Gates considers all of the quality metrics for a project and assigns a passed or failed designation for that project. It is possible to set a default Quality Gate which will be applied to all projects not explicitly assigned to some other gate.

- Click on the **Quality Gates** Tab



3) What are quality Profiles in sonarqube ??

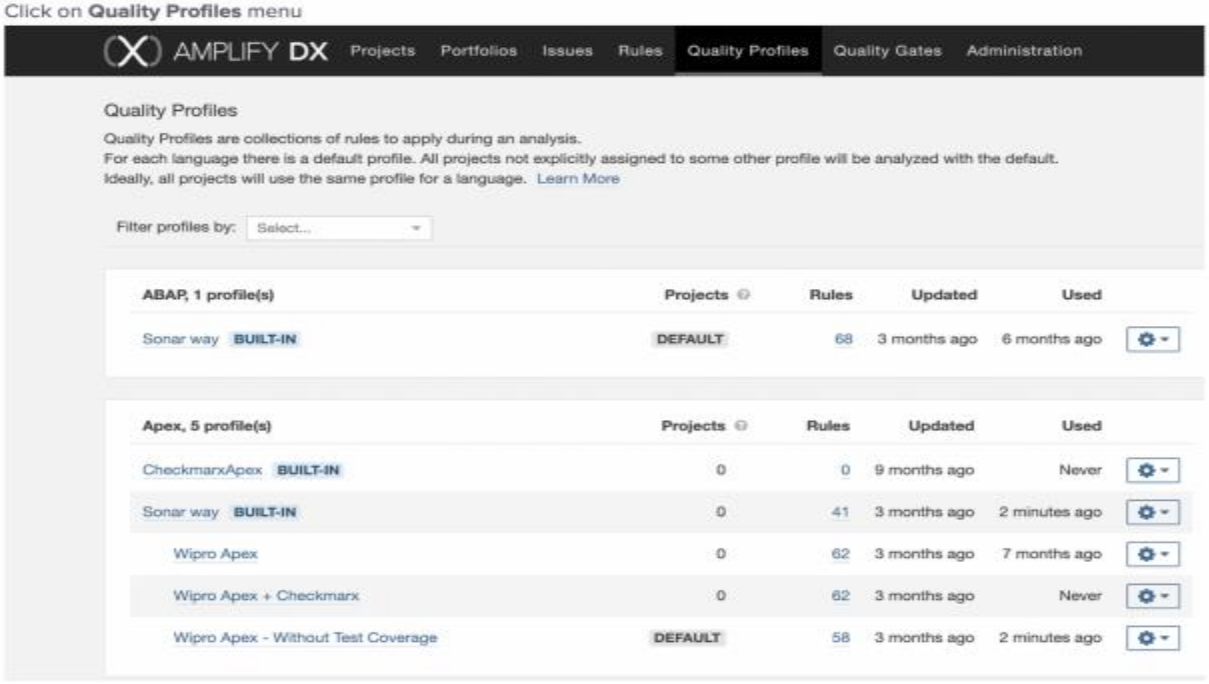
1) Quality Profiles are core component of SonarQube where you define sets of rules.
2) Each individual language has its own quality profile.

Path

1) Open your project in SonarQube.
2) Go to the Administration > Quality Profile menu.
3) Choose the quality profile you want to use for each language.

Languages are c, c++, Java, HTML, go many more.

- Click on **Quality Profiles** menu

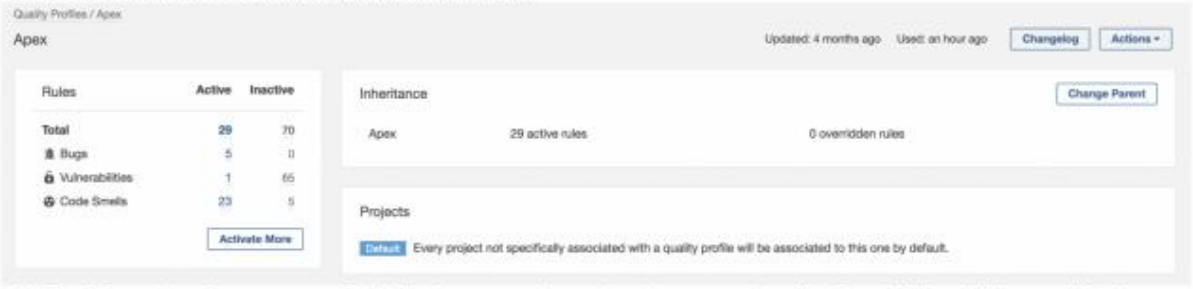


4) Which quality gate and quality profiles are managed in your organisation ?

We have managed the default quality gates and quality profiles in my organisations.

we have not tweaked any quality gates or quality profiles.

- Click on any profile to see detailed information like this



5) Code coverage

Code coverage, also called test coverage, is a measure of how much of the application's code has been run in testing.
Test coverage reports will update what percentage of the code is covered by each test cases.

6) Which library you have used for the code coverage evaluation ?

SonarQube is used in integration with JaCoCo, a free code coverage library for Java.

7) Scanner Sonar-scanner is the plugin required to integrate the sonar-tube with Jenkins the SonarScanner for Maven is recommended as the default scanner for Maven projects.

8) Which port Sonar uses ??

9000

---

9) Day to Day Activity

1) Will check any High Priority Emails, if we have High Priority Emails then will start looking at Prometheus and Grafana monitoring tools (will check logs why its failing), If no issue observed.
2) Later will check Teams if any issue reported by the developers team for any CI CD failing or any job not building, deploying or Jenkins worker node is not working.
3) I will look at my Jira board (Ticketing Tool) for planned tasks on the current sprints (creating CI CD, Building Terraform script, doing automation and adding Sonarqube).
4) We will have daily meetings called Standup (Updates on the tickets), 2 weeks once we will do sprints meeting for upcoming sprints.
5) I will try to do some automation (Shell Scripting), which we are doing manually.

As Devops Engineer

A) I will build an infrastructure using Terraform
B) I will configure infrastructure using Ansible
C) I will Deploy applications using Jenkins
D) I will CICD from the scratch
E) I have written CICD with Multiple stages
F) I will run the Sonar cube to test quality of the source code
G) Merging the Branches, creating a release branch etc

Kubernetes Architecture:
- Kubernetes components:
- API Server
- The API server is the main management point of the entire cluster.
- All the components in the cluster communicates through API server.
- Whenever we run kubectl command, we are actually communicating with the master's API server
- This is the only component which communicates with the etcd.
- The cluster authentication and authorization mechanism happens in API server
- API server has got watching mechanism for watching changes in the cluster.

- etcd:
- /etc distributed
- Its a distributed key value store which is used for configuration management, service discovery and all information about the cluster.
- Its default port is 2379
- It can be used to perform a variety of actions such as
* set, update remove keys and its values
* verify cluster health
* Generates DB snapshots

- Scheduler:
- The scheduler always watches for unscheduled jobs/pods (New pod request)
- According to the availability of required resources, quality of service requirements, affinity and anti-affinity configuration, scheduler will decide which node the pod should and should not be created.
- Once the pod has node assigned by the scheduler, the regular behaviour will continue.

- Kubernetes worker nodes:
- kubelet:
- kubelet is an agent that runs on each worker node and is responsible for watching API server for pods related changes that are bound to that particular node.
- Kubelet is the one which makes sure the pods are running.
- kubelet also communicates to docker daemon using API over the docker socket.

Some of the responsibilities of kubelet are:
- Run the containers in pod or keep the pod running
- Always report the status of the node and each pod to the API server
- Run container probes
- Monitors and collects the log data using CAdvisor

-Types of pod failures:
1. CrashLoopBackoff
2. ImagePullBackoff
3. FailedContainer

- kube proxy:
- It means that its a service. A service in k8s means a network

- kube proxy runs on each node and is responsible for watching API server for changes in services.
- Based on the pod definitions it manages the entire network configuration.
- proxying happens using pod_name.svc.<namespace>.cluster.local:<port_number>

- pod:
- Pods are the smallest deployable units of the k8s
- Each pod contain n number of containers and minimum of 1 container.
- If pod contains more than one container then these containers will share the same storage and network resources assigned to the pod.

Assignment:
Detailed view on Monolithic Applications and microservices.
* Get notes on Stateful and Stateless Applications
* Monolithic Applications and MicroServices
* What is Orchestration in IT industry

Stateful Applications:
- Stateful applications saves the user session data at the server side
- If servers get down it is difficult to transfer the user sessions
- This type wont work if we want to auto-scale our application servers.

Stateless Applications:
- Stateless application does not store any user data at server side.
- Using a single authentication gateway or client token methodology, data can be stored to validate the users once for multiple microservices.

- k8s services:
- In k8s, service is a REST API object. Set of rules/policies of accessing set of pods.

```
apiVersion: v1
kind: Service
metadata:
  name: headless-svc
spec:
  clusterIP: None
  selector:
    app: web
  ports:
    - protocol: TCP
      port: 80
           targetPort: 8080
```

Types of services:
Cluster_IP
- This is a default type of service which exposes the pod IP to other pods.
- We can reach the service cluster IP only within the k8s cluster.

- Its a way of assigning IP address and a kube DNS to set of pods.

- NodePort
- A NodePort service is the most primitive way to get the external traffic directed to our service or application running inside the cluster.
- The default LoadBalancer of k8s is NodePort
- This exposes the service on each node IP at a static port.
- Automatically a cluster_IP will be created internally

Cluster_IP
port_mapping_to_host = NodePort
- This service is used to expose the pod outside the cluster.
- If we want to specify any port while defining the NodePort k8s will automaticlly assign ports ranging between 30000-32767
- NodePort by default opens the specified port in all worker nodes in the cluster

-LoadBalancer
- This is a type of service used to link external LoadBalancer to the cluster.
- This is typically implemented by a cloud service provider and mainly depends on cloud services.
- A network LoadBalancer with an IP address is used to access the service.

- Ingress
What is ingress and why we need ingress?
- k8s ingress is an API object that provides routing rules to manage access to the services/microservices within the k8s cluster.
- This typically uses http/https protocol to facilitate the routing.

Some usecases of ingress include:
- Providing externally reachable URLs for services
- LoadBalancing traffic
- Offering Name based virtual hosting
- Terminating SSL

Ingress API object:
- The API object indicates the services that need to be exposed outside the cluster. It consists of routing rules.

Ingress Controller:
- Ingress controller is the actual implementaion of Ingress. Its usually a LoadBalancer that routes traffic from the API to the desired services/microservices within the k8s cluster.
OSI model: https://www.geeksforgeeks.org/layers-of-osi-model/

Service Discovery
- There are 2 ways to discover a service
DNS (This is a recommended method)
- The DNS server is added to the cluster in order to watch the k8s API create DNS record sets for each new service.

ENV var

- Pod runs on node, so that the k8s adds environment variables for each active service (each active pod)
pod_name.svc.namespace.cluster.local:32767

Headless Services:
- When we neither need not want load-balncing and any IP association, we create headless service by specifying none for clusterIP in the manifest file.

Controllers / kube-controller manager
- Controllers are the control loops which will be always running and always tries to match the current state of the cluster to the desired state which we specify in manifest.yaml
- Each controller is a seperate process, but to reduce complexity they are kept in compiled in single binary and run in a single process.

Default controller
Node controller
- Looks for node down and responds to API server when a node is down.
Endpoint controller
- It populates the information of endpoint objects (pods, jobs, cronjobs, services etc)
Deployment controller
- A deployment controller provides updates for pods and its replicaset
- It describes the desired state in deployment and the deployment controller changes and maintains the desired state
- Creates a deployment to rollout a replicaset
- Rollback to an earlier deployment revision.
- Scale up the deployment to match the load-balncing
- Using deployment controller, we can pause the deployment and we can apply the changes and fixes
- There are respective deployment strategies, we can specify it it manifest file and Deployment controller will take care of it.

Here, after this, I need you to have a glance on
* Different Deployment strategies which we use in k8s deployment
* Difference between deployment.yaml and pod.yaml
* What are the advantages of using deployment.yaml

Labels and Selectors:
- Labels are key-value pairs that are attached to objects. such as pods, nodes etc.
- In k8s, we can have multiple lables for a single object.
- Labels key must be identical where are value can be duplicated.
- We've got k8s predefined labels also.

Selectors (Label selectors)
- Using a selector,the client/user can identify a set of objects.
- The label selector is the core/primary way of grouping in k8s.

Equity based selectors
- We can only check the single value of label.

- Three kinds of operators are accepted. =, == and !=
Set Based Selectors
- Filter the keys according to set of values.
- Three kinds of operators are accepted/supported. in, notin and exists.
ex: environments in (production, qa)

Replicaset:
- A replica set ensures that a specified number of pod replicas are running at any given point of time.
- Uses set-based selectors for selection of objects such as pods
- Deployment always uses replicaset internally and its better to use deployment object instead of using Replicaset object directly.
- Deployment is higher level object that compared to replicaset.

Replication controller:
- A replication controller ensures that a specified number of pod replicas are up and running at a given point of time.
- It uses equity-based selectors to achieve desired state of replicas.
- Its an old way of replicating the pods.

Daemonset:
- A daemonset runs a copy of pod in all the nodes of cluster.
- It ensures that all node run that copy of pod which is created by daemonset.
example: running logs collection daemon/pod on every node running a node monitoring daemon/pod on every node.

Cronjob:
- Cronjob is used for creating periodic and recurring tasks. like running backups, sendint emails etc
- Cronjob can also schedule individual tasks for a specific time.

job:
- A job creates one or more pods and ensures that a specified number of pods are terminated succesfully
- The job can track the successful completion of pod execution
- We can run a job in different ways also
* one-time, sequential and parallel

Probes:
Liveness Probe and Readiness Probe

Readiness probe:
- Readiness probe is defined to check whether the application in the newly created pod is ready to accept the triffic.
- initialDelaySeconds is delay that k8s waits for those many seconds before connecting to the pod.
- periodSeconds is defined to check periodically whether the pod is ready to accept the traffic.

Liveness probe:
- Liveness is used to verify if the application is up and running (alive) correctly or does it need to restart.

How do we create a pod in a particular node?
- nodeSelector
- We need to define a label to node
- We can define nodeselector in pod spec section in .yaml file so that the pod is created in that particular node
- If we have more than one node with same label then pod will be created based on the load (scheduler will decide)

- affinity and anti-affinity
- We can use node-affinity to assign a pod to particular node but in our company, we are using nodeSelector.

How do we write .yaml files

```
---
apiVersion: v1
kind: pod
metadata:
      label:
            my-node
spec:
 container:
     Image:
          my-image
     port:
       80

---
apiVersion: v1
kind: deployment
metadata:
       label:
            my-node
spec:
   Image:
      my-image
            tag:
         0.0.1

 ---

   apiVersion: v1
   kind: Service
   metadata:
       name: headless-svc
    spec:
```

```
    clusterIP: None
     selector:
       app: web
      ports:
       - protocol: TCP
         port: 80

           targetPort: 8080
```

Setup a K8s cluster on Windows:
https://kubernetes.io/docs/tasks/tools/install-kubectl-windows/

ConfigMaps
- A configmap is an API object used to store non-confidential data in key-value pairs
- Pods can consume ConfigMaps as Environment Variables, Command-line argument, or as configuration files in a volume.

cm.yaml

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: game-demo
data:
  username: ZGV2b3A=
  password: QWRtaW5AMTI=

username: $username

---
```

Secrets
- Secret is an API object used to store confidential data in key-value pairs
- It is pod specific. No other pod will be able to read the secret value unless and untill specified in .yaml explicitly.

secret.yaml

```
apiVersion:
kind: secret
metadata:

data:
```

Network Policies:
- If we want to control traffic flow at the IP address or port level (OSI layer 3 or 4), then we might consider using NetworkPolicies.
- prerequisites of network policies are,
- network plugin
- networking solution which supports network policy

- Creating a NetworkPolicy resource without a controller that implements it will have no effect.

The below topics needs a little bit of explanation.

Namespaces:
deploying EKS cluster on AWS
LoadBalancing:
Istio
helm charts.

Commands in k8s:

- To list the nodes:
kubectl get nodes

- To Create namespace
kubectl create ns <namespace_name>

- To list the namespaces:
kubectl get ns

- To list the pods in All namespaces:
kubectl get pod -A

- To list the pod in specific name space:
kubectl get pod -n namspace_name

- To list services, persistant volume, configmap, secret
kubectl get svc,pvc,cm,secret -n namespace_name

- To check the logs
kubectl logs -f pod/pod_name -c container_name

- To describe the pod:
kubectl describe pod/pod_name -n namespace_name

- To list the images:
kubectl describe deploy -n namespace_name | grep -i image

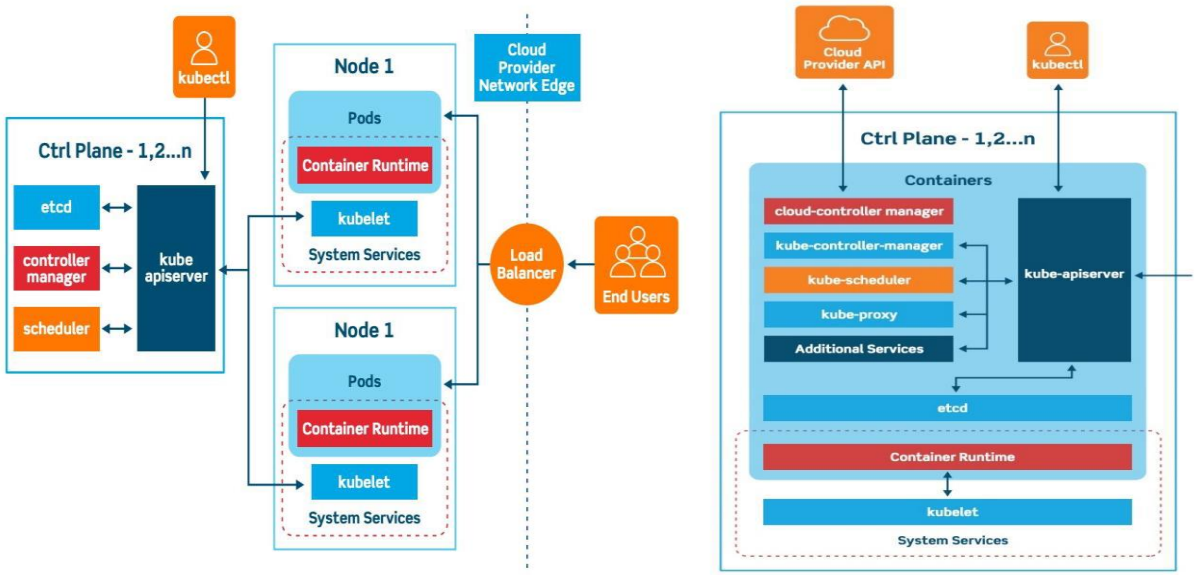
- To edit any .yaml,
kubectl edit <kind> <kind_label> -n namespace_name
kind can be a pod, deployment, job, service, ingress etc
kind_label is the name givne to that particular yaml

- To rollback
kubectl rollout undo deploy deployment_name -n namespace_name

- To get into the pod/container
kubectl exec -it pod/pod_name -c container_name -n namespace_name /bin/bash

1) Kubernetes
Version – I worked from 1.18 to 1.22
kubernetes is written in **go language**
Terraform is written in **go language**

Kubernetes is an open source container orchestration tool developed by google to help you manage the containerised / dockerised application supporting multiple deployment environments like on premise, cloud, or virtual machines.
What are the main features of K8s offers?
1) Assures high availability with zero down time
2) High performance and scalable
3) Reliable infrastructure to support data recovery
Disadvantages
It is very complex or takes lot of time for setting up the k8s infra
---
2) K8s architecture or cluster:



1) K8s cluster has two major components
1) Master node
2) Worker node
2) Node
physical machine or virtual worker machine where containers will be deployed by k8s.
3) Cluster
set of nodes grouped together.
4) Components of K8s
API server, scheduler, control manager, etcd, kubelet, container runtime and kube-proxy.
Master or control panel manages the cluster

1) Kubernetes Architecture
Kubernetes has two nodes – master node and worker/slave node.

1) Master node

The master node is the first and most vital component which is responsible for the management of Kubernetes cluster. it is the entry point of all kind of administrative tasks. there might be more than one master node in the cluster to check for fault tolerance.

The master node has various components like API server, Controller manager, kubectl, scheduler and ETCD.

A) ETCD

1) It is the Meta data storage (It will stored in Json format(Keyvalue)). Configuration details and essential values in key value form

2) Complete kubanetes information will be available in ETCD. It will have the full information about which container is running, which connector is connected and other information's

3) Cluster information's will be stored in ETCD.

4) Its default port is 2379

{
"name": "xyz",
"part": "ght"
}

B) Controller Manager:

1) Control Manager is component, runs controllers to administer nodes and endpoints.

2) Runs in continuous loop and is responsible for gathering information and sending it to the API server.

C) Scheduler

1) The Scheduler assigns the tasks to the slave nodes.

2) It is responsible for distributing the workload and stores resource usage information on every node.

D) API Server

1) API Server is central management entity.

2) All the components in the cluster communicate through API server.

3) Any information for Kubectl to be fetched or to deploy from API Server, Kubectl will convert in form of API Lanuguage (REST API).

4) This is the only component which communicates with the etcd.

E) Kubectl

1) Kubectl is the component controls and communicates with kubernetes cluster.

Note: All Master Node components will be running as a POD

2) Worker/Slave nodes
Worker nodes are another essential component, manage the networking between the containers, communicate with the master node, it will assign resources to the containers.
A) Kubelet: Communication between Master and Worker node. Kubelet is the one which makes sure the pods are running.
B) Docker Container: Docker container runs on each of the worker nodes, which will run to configure pods.
C) Kube-proxy: Kube-proxy acts as a load balancer and network proxy to perform service on a single worker node.
D) Pods: A pod is a combination of single or multiple containers that logically runs together on nodes.
Pods:
Pods are smallest objects which can be created in Kubernetes cluster.
In terms of docker concepts, a pod is similar to a group of docker containers with shared namespace and shared file system volumes.

Q) what is overlay network using in Kubernetes
Answer: Calico
---
3) Kubernetes Basics
1) Cluster
It is set of nodes, which grouped together, that include ram, CPU, disk, and their devices.
---
4) Kubernetes Master Setup
sudo su -
2) # upgrade & install depedencies
a) apt update && apt upgrade -y && apt dist-upgrade -y
b) apt install -y apt-transport-https ca-certificates curl software-properties-common gnupg2 net-tools

3) # set FQDN hostname
a) hostnamectl set-hostname $(curl -s http://169.254.169.254/latest/meta-data/local-hostname)

4) # install docker
a)
apt install -y docker.io
cat << EOF > /etc/docker/daemon.json
{
  "exec-opts": ["native.cgroupdriver=systemd"],
  "log-driver": "json-file",
  "log-opts": {
    "max-size": "100m"
  },
  "storage-driver": "overlay2"
}

EOF

b)
1) systemctl stop docker
2) systemctl start docker
3) systemctl enable docker

c)
cat <<EOF > /etc/sysctl.d/k8s.conf
net.bridge.bridge-nf-call-ip6tables = 1
net.bridge.bridge-nf-call-iptables = 1
net.ipv4.ip_forward = 1
EOF
d)
sysctl --system

5) # install kubelet kubeadm kubectl
a)
1) curl -s https://packages.cloud.google.com/apt/doc/apt-key.gpg | sudo apt-key add
2) apt-add-repository "deb http://apt.kubernetes.io/ kubernetes-xenial main"
3) apt update
4) apt install kubelet="1.22.17-00"
5) apt install kubeadm="1.22.17-00"
6) apt install kubectl="1.22.17-00"

b)
cat << EOF > /etc/kubernetes/aws.yaml
---
apiVersion: kubeadm.k8s.io/v1beta2
kind: ClusterConfiguration
networking:
  serviceSubnet: "10.100.0.0/16"
  podSubnet: "10.244.0.0/16"
apiServer:
  extraArgs:
    cloud-provider: "aws"
controllerManager:
  extraArgs:
    cloud-provider: "aws"
EOF

6) # init cluster
1) kubeadm init --config /etc/kubernetes/aws.yaml

The below mentioned command generated from system (Copy for future)
kubeadm join 10.0.0.31:6443 --token 9j7g49.d0xttj11jt8nbgkl \
        --discovery-token-ca-cert-hash
sha256:d6ef195132a18a988046b4b5ee8d93ff20a92e499ee1b05278bace64f73e6e61
(Save this details. It is used in node.yaml to connect Control plane)

7) # save kube config
1) mkdir -p $HOME/.kube
2) sudo cp -i /etc/kubernetes/admin.conf $HOME/.kube/config
3) sudo chown $(id -u):$(id -g) $HOME/.kube/config

8) # install Calico
1) kubectl apply -f https://docs.projectcalico.org/manifests/calico-typha.yaml
2) kubectl get nodes

Kubernetes Worker Node Setup
sudo su -
A) # upgrade & install depedencies
1) apt update && apt upgrade -y && apt dist-upgrade -y
2) apt install -y apt-transport-https ca-certificates curl software-properties-common gnupg2 net-tools

2) # set FQDN hostname
1) hostnamectl set-hostname $(curl -s http://169.254.169.254/latest/meta-data/local-hostname)

3) # install docker
a)
```
apt install -y docker.io
cat << EOF > /etc/docker/daemon.json
{
  "exec-opts": ["native.cgroupdriver=systemd"],
  "log-driver": "json-file",
  "log-opts": {
    "max-size": "100m"
  },
  "storage-driver": "overlay2"
}
EOF
```

b)
1) systemctl stop docker
2) systemctl start docker
3) systemctl enable docker

c)
```
cat <<EOF > /etc/sysctl.d/k8s.conf
net.bridge.bridge-nf-call-ip6tables = 1
net.bridge.bridge-nf-call-iptables = 1
net.ipv4.ip_forward = 1
EOF
sysctl --system
```

4) # install kubelet kubeadm kubectl

a)
1) curl -s https://packages.cloud.google.com/apt/doc/apt-key.gpg | sudo apt-key add
2) apt-add-repository "deb http://apt.kubernetes.io/ kubernetes-xenial main"
3) apt update
4) apt install kubelet="1.22.17-00"
5) apt install kubeadm="1.22.17-00"
6) apt install kubectl="1.22.17-00"

b)
```
cat << EOF > /etc/kubernetes/node.yml
---
apiVersion: kubeadm.k8s.io/v1beta1
kind: JoinConfiguration
discovery:
  bootstrapToken:
    token: "9j7g49.d0xttj11jt8nbgkl"
    apiServerEndpoint: "10.0.0.31:6443"
    caCertHashes:
      -
"sha256:d6ef195132a18a988046b4b5ee8d93ff20a92e499ee1b05278bace64f73e6e61"
nodeRegistration:
  name: ip-10-0-0-186.eu-west-3.compute.internal
  kubeletExtraArgs:
    cloud-provider: aws
EOF
```

Change in above commands with
1) Token
2) apiServerEndpoint
3) caCertHashes
4) nodeRegistration
Note: command hostname –f = to get ip-10-0-0-186.eu-west-3.compute.internal
(Worker node)

1) kubeadm join 10.0.0.31:6443 --token 9j7g49.d0xttj11jt8nbgkl \
--discovery-token-ca-cert-hash
sha256:d6ef195132a18a988046b4b5ee8d93ff20a92e499ee1b05278bace64f73e6e61

5) #join Cluster
kubeadm join --config /etc/kubernetes/node.yml

---

5) Commands
1) To check pods are running on master or worker node
Command: kubectl get pods -A -owide

```
root@ip-10-0-0-31:~# kubectl get pods -A -owide
NAMESPACE     NAME                                       READY   STATUS    RESTARTS       AGE     IP              NODE
                                NOMINATED NODE   READINESS GATES
kube-system   calico-kube-controllers-68d86f8988-hfrls   1/1     Running   2 (44m ago)    22h     10.244.24.137   ip-10
-0-0-31.ap-south-1.compute.internal    <none>           <none>
kube-system   calico-node-hthzd                          1/1     Running   1 (44m ago)    5h29m   10.0.0.122      ip-10
-0-0-122.ap-south-1.compute.internal   <none>           <none>
kube-system   calico-node-lb5vs                          1/1     Running   1 (44m ago)    5h35m   10.0.0.95       ip-10
-0-0-95.ap-south-1.compute.internal    <none>           <none>
```

2) To check worker nodes connected
Command: kubectl get nodes

```
root@ip-10-0-0-31:~# kubectl get nodes
NAME                                        STATUS   ROLES                  AGE     VERSION
ip-10-0-0-122.ap-south-1.compute.internal   Ready    <none>                 5h29m   v1.22.17
ip-10-0-0-31.ap-south-1.compute.internal    Ready    control-plane,master   22h     v1.22.17
ip-10-0-0-95.ap-south-1.compute.internal    Ready    <none>                 5h36m   v1.22.17
```

3) Particular namespace pods
Command: kubectl get pods -n kube-system

```
root@ip-10-0-0-31:~# kubectl get pods -n kube-system
NAME                                                              READY   STATUS    RESTARTS        AGE
calico-kube-controllers-68d86f8988-hfrls                          1/1     Running   2 (46m ago)     22h
calico-node-hthzd                                                 1/1     Running   1 (46m ago)     5h30m
calico-node-lb5vs                                                 1/1     Running   1 (46m ago)     5h37m
calico-node-nhdg9                                                 1/1     Running   52 (46m ago)    22h
calico-typha-54b5d5df68-vjftl                                     1/1     Running   1 (46m ago)     22h
coredns-7bdbbf6bf5-7qx7s                                          1/1     Running   2 (46m ago)     22h
coredns-7bdbbf6bf5-hjfd6                                          1/1     Running   2 (46m ago)     22h
etcd-ip-10-0-0-31.ap-south-1.compute.internal                     1/1     Running   2 (46m ago)     22h
kube-apiserver-ip-10-0-0-31.ap-south-1.compute.internal           1/1     Running   2 (46m ago)     22h
kube-controller-manager-ip-10-0-0-31.ap-south-1.compute.internal  1/1     Running   2 (46m ago)     22h
kube-proxy-2jgn4                                                  1/1     Running   1 (46m ago)     5h30m
kube-proxy-6wxwh                                                  1/1     Running   1 (46m ago)     5h37m
kube-proxy-wkqnx                                                  1/1     Running   2 (46m ago)     22h
kube-scheduler-ip-10-0-0-31.ap-south-1.compute.internal           1/1     Running   2 (46m ago)     22h
```

4) Command: kubectl get pods -n kube-system –owide

```
root@ip-10-0-0-31:~# kubectl get pods -n kube-system -owide
NAME                                       READY   STATUS    RESTARTS      AGE     IP              NODE
                    NOMINATED NODE   READINESS GATES
calico-kube-controllers-68d86f8988-hfrls   1/1     Running   2 (48m ago)   22h     10.244.24.137   ip-10-0-0-31.ap-sou
th-1.compute.internal    <none>           <none>
calico-node-hthzd                          1/1     Running   1 (48m ago)   5h32m   10.0.0.122      ip-10-0-0-122.ap-so
uth-1.compute.internal   <none>           <none>
calico-node-lb5vs                          1/1     Running   1 (48m ago)   5h39m   10.0.0.95       ip-10-0-0-95.ap-sou
th-1.compute.internal    <none>           <none>
```

---

6) Namespaces:

1) Namespace helps in creating a virtual cluster inside Kubernetes cluster.
2) Namespace helps you isolate resources between different teams.
3) It is a logical cluster or environment. It is a widely used method which is used for dividing a cluster.

Advantages of namespace:

1) Control multiple app with single name in a single cluster
2) Helps in easy management of applications of each environment
3) Cost effective

Command: kubectl get namespace **or** kubectl get ns

```
root@ip-10-0-0-31:~# kubectl get namespace
NAME              STATUS   AGE
default           Active   22h
kube-node-lease   Active   22h
kube-public       Active   22h
kube-system       Active   22h
root@ip-10-0-0-31:~# kubectl get ns
NAME              STATUS   AGE
default           Active   22h
kube-node-lease   Active   22h
kube-public       Active   22h
kube-system       Active   22h
```

1) kube-node-lease
2) kube-public
3) kube-system
all this are related to Control plane, we cannot deploy anything

Note: To create any container. We will be using **default namespace**

1) Default: adding an object to a cluster without providing a namespace will place it within the default namespace
2) Kube-system: kube system namespace is used for k8s components managed by k8s
3) Kube-public: kube public it is readable by everyone, but the namespace is reserved for system usage

---

7) YAML (yet another mark-up language) File:
YAML is a human readable data serialization language. It is commonly used for configuration files and in applications where data is being stored or
Command: Kubectl create -f filename

Life cycle of k8s:
1) Pending: if the container is not neither in running or terminated state it is pending.
2) Running: the running status indicates the container is running without issues
3) Terminated: the container in the terminated began execution and then either run to completion or fails for some reason.

Steps
Create a file with **vi myfirstpod.yaml**

```
apiVersion: v1
kind: Pod
metadata:
  name: tomcat-nginx
  labels:
    app: tomcat
spec:
  containers:
  - name: mytomcat
    image: yetish519/mytomcat 2.0
  - name: nginx
    image: nginx:1.14.2
```

1) Command to Execute: kubectl create -f myfirstpod.yaml **OR** kubectl apply -f myfirstpod.yaml

```
root@ip-10-0-0-31:~# kubectl create -f myfirstpod.yaml
pod/tomcat-jenkins created
```

2) Command: kubectl get pods

```
root@ip-10-0-0-31:~# kubectl get pods
NAME             READY   STATUS             RESTARTS   AGE
tomcat-jenkins   1/2     ImagePullBackOff   0          4m44s
```

3) later check in worker node, tomcat will be running **(Command: docker ps)**

```
root@ip-10-0-0-122:~# docker ps
CONTAINER ID   IMAGE                  COMMAND                  CREATED          STATUS          PORTS     NAMES
a6aff7c245d3   nginx                  "nginx -g 'daemon of…"   6 minutes ago    Up 6 minutes              k8s_myjenkins_tomcat-jenki
ns_default_219869d4-e51a-4d82-aeb5-f01f1be165f0_0
704a9cd44276   registry.k8s.io/pause:3.5   "/pause"                 6 minutes ago    Up 6 minutes              k8s_POD_tomcat-jenkins_def
ault_219869d4-e51a-4d82-aeb5-f01f1be165f0_0
```

4) Command: kubectl get pods –owide – **To check, where container it is running (worker node)**

```
root@ip-10-0-0-31:~# kubectl get pods -owide
NAME            READY   STATUS             RESTARTS   AGE   IP              NODE                                          NOMINATED NODE   READ
INESS GATES
tomcat-jenkins   1/2     ImagePullBackOff   0          14m   10.244.146.193   ip-10-0-0-122.ap-south-1.compute.internal   <none>           <non
e>
root@ip-10-0-0-31:~# curl -v 10.244.146.193
```

5)

a) Command curl –v 10.244.146.193:8080 **(Tomcat)**

b) Command curl –v 10.244.146.193:80 **(nginx)**

6) To delete the pod

Command: kubectl delete pod tomcat-jenkins **= to delete the pods**

```
root@ip-10-0-0-31:~# kubectl delete pod tomcat-jenkins
pod "tomcat-jenkins" deleted
```

7) Command: kubectl get services **= To check the services**

```
root@ip-10-0-0-31:~# kubectl get services
NAME          TYPE         CLUSTER-IP    EXTERNAL-IP   PORT(S)   AGE
kubernetes    ClusterIP    10.100.0.1    <none>        443/TCP   23h
```

8) Command: To delete the service

1) kubectl get services

2) kubectl delete service tomcat-nginx

```
root@ip-10-0-0-31:~# kubectl delete service tomcat-nginx
service "tomcat-nginx" deleted
```

Steps Explanation

1) apiVersion: v1 = Convert to API call

2) kind: Pod = What type of object, we are creating

3) metadata:

name: tomcat- nginx

To give the name for pod

4) labels:

app: tomcat

key value

5) spec:

containers:

- name: mytomcat

image: yetish519/mytomcat

- name: nginx

image: nginx:1.14.2

Specifications for creating a Pod

---

8) Publishing Services

K8s service type allows you to specify what kind of service you want, the default is cluster IP

Type values and their behaviors are

1) ClusterIP: Exposes the service on a cluster internal IP. Using this value makes the service only reachable from within the cluster. This is the default service type.

2) NodePort: exposes the service on each nodes IP at a static port.

A cluster IP service, to which the nodes port service routes are automatically created.

3) LoadBalancer: exposes the service externally using a cloud provider's load balancer.

4) ExternalName/ExternalIP: maps the service to the content of external fields by written a C name records within a value.

Note: Node port and cluster IP services, to which the external load balancer routes or automatically created.

1) ClusterIP

Function:

1) If want the access services or pods with the cluster

```
apiVersion: v1
kind: Service
metadata:
  name: tomcat-nginx1
spec:
  selector:
    app: tomcat
  type: ClusterIP
  ports:
    - name: tomcat
      protocol: TCP
      targetPort: 8080
      port: 8080
    - name: nginx
      protocol: TCP
      targetPort: 80
      port: 80
```

1) Below command **app: tomcat** should match the previously created on **myfirstpod.yaml**

```
spec:
  selector:
    app: tomcat
```

2) Service

```
type: ClusterIP
```

Commands:

1) kubectl apply -f myfirstpod.yaml

```
root@ip-10-0-0-31:~# kubectl apply -f service.yaml
service/tomcat-nginx1 created
```

2) kubectl get services

```
root@ip-10-0-0-31:~# kubectl get services
NAME             TYPE        CLUSTER-IP      EXTERNAL-IP   PORT(S)            AGE
kubernetes       ClusterIP   10.100.0.1      <none>        443/TCP            45h
tomcat-nginx1    ClusterIP   10.100.13.71    <none>        8080/TCP,80/TCP    23s
```

3) curl -v 10.100.13.71:8080

```
root@ip-10-0-0-31:~# curl -v 10.100.44.1:8080
*   Trying 10.100.44.1:8080...
* Connected to 10.100.44.1 (10.100.44.1) port 8080 (#0)
> GET / HTTP/1.1
> Host: 10.100.44.1:8080
> User-Agent: curl/7.81.0
> Accept: */*
>
* Mark bundle as not supporting multiuse
< HTTP/1.1 404
< Content-Type: text/html;charset=utf-8
< Content-Language: en
< Content-Length: 682
< Date: Fri, 20 Jan 2023 08:27:52 GMT
<
* Connection #0 to host 10.100.44.1 left intact
<!doctype html><html lang="en"><head><title>HTTP Status 404 – Not Found</title><style type="text/css">body {font-family:Tahoma,Arial,sans-serif
;} h1, h2, h3, b {color:white;background-color:#525D76;} h1 {font-size:22px;} h2 {font-size:16px;} h3 {font-size:14px;} p {font-size:12px;} a {
color:black;} .line {height:1px;background-color:#525D76;border:none;}</style></head><body><h1>HTTP Status 404 – Not Found</h1><hr class="line"
 /><p><b>Type</b> Status Report</p><p><b>Description</b> The origin server did not find a current representation for the target resource or is
not willing to disclose that one exists.</p><hr class="line" /><h3>Apache Tomcat/9.0.70</h3></body></html>root@ip-10-0-0-31:~#
```

4) kubectl get endpoints

```
root@ip-10-0-0-31:~# kubectl get endpoints
NAME             ENDPOINTS                                      AGE
kubernetes       10.0.0.31:6443                                 46h
tomcat-nginx     10.244.146.196:8080,10.244.146.196:80          3m32s
```

5) kubectl get pods --show-labels **(To check label)**

```
root@ip-10-0-0-31:~# kubectl get pods --show-labels
NAME             READY   STATUS    RESTARTS   AGE   LABELS
tomcat-nginx     2/2     Running   0          75m   app=tomcat
tomcat-nginx1    2/2     Running   0          43s   app=tomcat
```

2) NodePort

Note: service-node-port-range (default: 30000-32767) port range

Function:

1) If want to access Pods from the outside of the cluster from my browser

```yaml
apiVersion: v1
kind: Service
metadata:
  name: tomcat-nodeport
spec:
  selector:
    app: tomcat
  type: NodePort
  ports:
    - name: tomcat
      protocol: TCP
      targetPort: 8080
      port: 8080
      nodeport: 30010
    - name: nginx
      protocol: TCP
      targetPort: 80
      port: 80
      nodeport: 30011
```

3) LoadBalancer

Function:

1) If want to load balance the traffic and continue working of my pods from outside browser

2) It will create the Load balancer in the AWS account and map node port of the master node or worker node

```yaml
apiVersion: v1
kind: Service
metadata:
  name: tomcat-loadbalancer
spec:
  selector:
    app: tomcat
  type: LoadBalancer
  ports:
  - name: tomcat
    protocol: TCP
    targetPort: 8080
    port: 8080
  - name: nginx
    protocol: TCP
    targetPort: 80
    port: 80
```

Commands:

1) kubectl apply -f loadbalancer.yaml

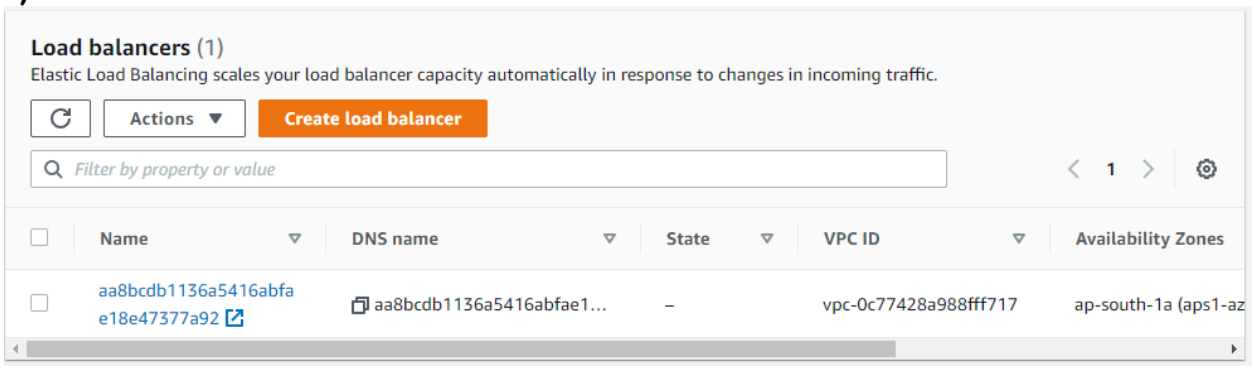
2) kubectl get service

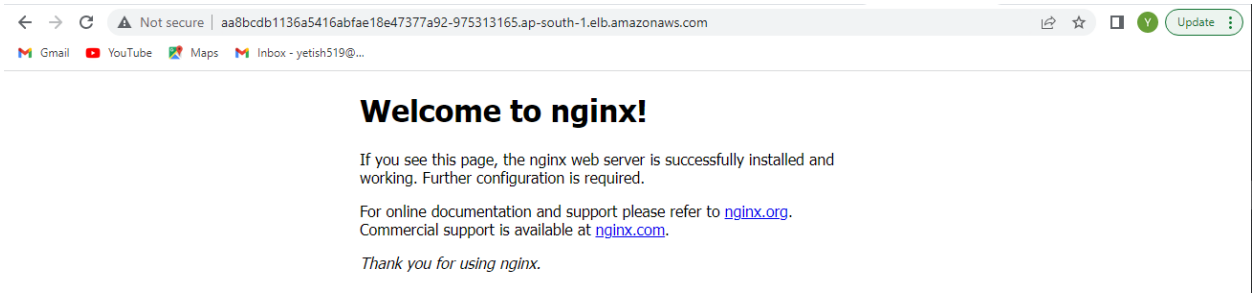
```
root@ip-10-0-0-31:~# kubectl get service
NAME                 TYPE           CLUSTER-IP      EXTERNAL-IP                                                               PORT(S)
          AGE
kubernetes           ClusterIP      10.100.0.1      <none>                                                                    443/TCP
          47h
tomcat-loadbalancer  LoadBalancer   10.100.159.75   aa8bcdb1136a5416abfae18e47377a92-975313165.ap-south-1.elb.amazonaws.com   8080:30329/TCP,8
0:32107/TCP   64s
tomcat-nginx         ClusterIP      10.100.44.1     <none>                                                                    8080/TCP,80/TCP
          32m
```

3) copy above link and hit in browser

http://aa8bcdb1136a5416abfae18e47377a92-975313165.ap-south-1.elb.amazonaws.com/

4) Check in LoadBalancer will created





4) ExternalName/ExternalIP

Function: External IP will be accessed can be used public IP of master

```yaml
apiVersion: v1
kind: Service
metadata:
  name: tomcat-external
spec:
  selector:
    app: tomcat
  ports:
  - name: tomcat
    protocol: TCP
    targetPort: 8080
    port: 8080
  - name: nginx
    protocol: TCP
    targetPort: 80
    port: 80
  externalIPs:
    - 43.205.110.236
```

43.205.110.236 – Public IP of Master Server

Commands

1) kubectl apply -f external.yaml

2) kubectl get service

```
root@ip-10-0-0-31:~# kubectl get service
NAME                  TYPE           CLUSTER-IP      EXTERNAL-IP                                                              PORT(S)
             AGE
kubernetes            ClusterIP      10.100.0.1      <none>                                                                   443/TCP
             47h
tomcat-external       ClusterIP      10.100.192.53   43.205.110.236                                                           8080/TCP,80/TCP
             5s
tomcat-loadbalancer   LoadBalancer   10.100.159.75   aa8bcdb1136a5416abfae18e47377a92-975313165.ap-south-1.elb.amazonaws.com   8080:30329/TCP,8
0:32107/TCP   19m
tomcat-nginx          ClusterIP      10.100.44.1     <none>                                                                   8080/TCP,80/TCP
             50m
```

---

9) ReplicaSet

1) A replica set ensures that a specified number of pod replicas are running at any given point of time.
2) Deployment always uses replicaset internally and its better to use deployment object instead of using Replicaset object directly.
3) Deployment is higher level object that compared to replicaset.

We want to replicate containers (and thereby applications) for several reasons, including:

1) Reliability: By having multiple versions of an application, you prevent problems if one or more fails. This is particularly true if the system replaces any containers that fail.
2) Load balancing: Having multiple versions of a container enables you to easily send traffic to different instances to prevent overloading of a single instance or node. This is something that Kubernetes does out of the box, making it extremely convenient.
3) Scaling: When load does become too much for the number of existing instances, Kubernetes enables you to easily scale up your application.

```yaml
apiVersion: apps/v1
kind: ReplicaSet
metadata:
  name: tomcat
spec:
  replicas: 4
  selector:
    matchLabels:
      app: tomcat
  template:
    metadata:
      name: tomcat-nginx
      labels:
        app: tomcat
    spec:
      containers:
        - name: tomcat
          image: prajwal1327/mytomcat:1.1.2
```

Commands:
1) kubectl get pods

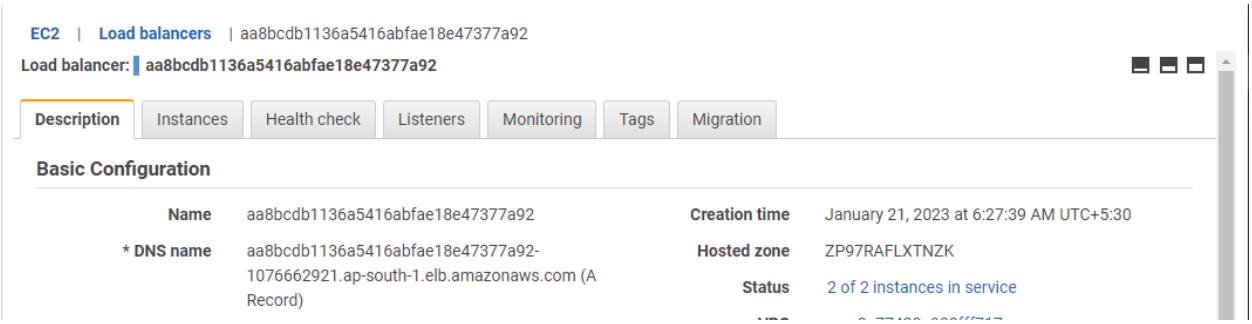
```
root@ip-10-0-0-31:~# kubectl get pods
NAME             READY   STATUS    RESTARTS   AGE
tomcat-jqjpw     1/1     Running   0          72s
tomcat-nginx1    2/2     Running   0          5m15s
tomcat-rnpfr     1/1     Running   0          4m28s
tomcat-znd9b     1/1     Running   0          72s
```

2) kubectl get replicaset

```
root@ip-10-0-0-31:~# kubectl get replicaset
NAME      DESIRED   CURRENT   READY   AGE
tomcat    4         4         4       3m33s
```

3) kubectl get pod tomcat-znd9b –oyaml

4) In Load balancer copy DNS name and hit in browser



5) http://aa8bcdb1136a5416abfae18e47377a92-1076662921.ap-south-1.elb.amazonaws.com:8080/hello-world-war-1.1.2/

6) To delete all pods
Command: kubectl delete pod –all

7) Command to replicaset
Command: kubectl delete replicaset tomcat

---

10) Deployment:

1) Deployment tells Kubernetes how to create or modify instances of the pods that hold a containerized application.
2) Without a deployment, we need to create, update, and delete a bunch of pods manually.
3) With a deployment, you declare a single object in a YAML file.

This object is responsible for creating the pods, making sure they stay up to date.

Rollingupdate:

1) Kubernetes deployments rollout pod version updates with a rolling update strategy.
2) This strategy aims to prevent application downtime by keeping at least some instance up and running at any point in time while performing the updates.
3) Old pods are only shutdown after new pods of the new deployment version have started up and became ready to handle the traffic.

1) MaxSurge: it indicates maximum number of pods need to be created before terminating the old pods.

2) maxUnavailable: this indicates number of unavailable pods during rolling updates.

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: tomcat
spec:
  replicas: 5
  selector:
   matchLabels:
     app: tomcat
  strategy:
    type: RollingUpdate
    rollingUpdate:
      maxSurge: 2
      maxUnavailable: 2
  template:
    metadata:
      name: tomcat-nginx
      labels:
        app: tomcat
    spec:
      containers:
        - name: tomcat
          image: krishna123456/tomcat:1.11.111
```

1) Only Strategy need to be added

```yaml
strategy:
  type: RollingUpdate
  rollingUpdate:
    maxSurge: 2
    maxUnavailable: 2
```

2) kubectl apply -f deployment.yaml

3) kubectl get deployments

```
root@ip-10-0-0-31:~# kubectl get deployments
NAME     READY   UP-TO-DATE   AVAILABLE   AGE
tomcat   5/5     5            5           9s
```

4) kubectl get replicaset

```
root@ip-10-0-0-31:~# kubectl get replicaset
NAME                DESIRED   CURRENT   READY   AGE
tomcat-944dbf64c    5         5         5       43s
```

5) kubectl get pods

```
root@ip-10-0-0-31:~# kubectl get pods
NAME                      READY   STATUS    RESTARTS   AGE
tomcat-944dbf64c-4tlmv    1/1     Running   0          56s
tomcat-944dbf64c-6qtgz    1/1     Running   0          56s
tomcat-944dbf64c-d6n24    1/1     Running   0          56s
tomcat-944dbf64c-lj5gj    1/1     Running   0          56s
tomcat-944dbf64c-xng78    1/1     Running   0          56s
```

---

11) Creating own names Space

Advantages of namespace:

1) Control multiple app with single name in a single cluster
2) Helps in easy management of applications of each environment
3) Cost effective

1)

```
apiVersion: v1
kind: Namespace
metadata:
  name: ys-namespace
```

Commands

1) kubectl get namespace

```
root@ip-10-0-0-31:~# kubectl get namespace
NAME              STATUS   AGE
default           Active   3d21h
kube-node-lease   Active   3d21h
kube-public       Active   3d21h
kube-system       Active   3d21h
ys-namespace      Active   21s
```

2) kubectl get pods -n ys-namespace = **To pods running under namespace which I created**

3) kubectl apply -f myfirstpod.yaml --namespace=ys-namespace

```
apiVersion: v1
kind: Pod
metadata:
  name: tomcat-nginx
  namespace: ys-namespace
  labels:
    app: tomcat
spec:
  containers:
  - name: mytomcat
    image: prajwal1327/mytomcat:1.1.6
 namespace: ys-namespace
```

Need to add namespace

4) kubectl get pods -n ys-namespace = **To check under my namespace**

```
root@ip-10-0-0-31:~# kubectl get pods -n ys-namespace
NAME           READY   STATUS    RESTARTS   AGE
tomcat-nginx   1/1     Running   0          68s
```

1) Resource Quote:
It will restrict the namespace for a particular CPU

```yaml
apiVersion: v1
kind: ResourceQuota
metadata:
  name: memory-cpu-quota
  namespace: ys-namespace
spec:
  hard:
    requests.cpu: "1"
    requests.memory: 1Gi
    limits.cpu: "2"
    limits.memory: 2Gi
```

Command:
1) kubectl apply -f devnames.yaml
2) kubectl describe namespace ys-namespace = **Describe is equal to inspect in docker**

```
root@ip-10-0-0-31:~# kubectl describe namespace ys-namespace
Name:         ys-namespace
Labels:       kubernetes.io/metadata.name=ys-namespace
Annotations:  <none>
Status:       Active

Resource Quotas
  Name:            memory-cpu-quota
  Resource         Used  Hard
  --------         ---   ---
  limits.cpu       0     2
  limits.memory    0     2Gi
  requests.cpu     0     1
  requests.memory  0     1Gi

No LimitRange resource.
```

Note:
1) Inspect in Docker
2) describe in K8s

To delete the resourcequota commands
1) kubectl get resourcequota -n ys-namespace
2) kubectl delete resourcequota memory-cpu-quota -n ys-namespace

2) LimitRange: any pod in the system will be able to consume as much CPU and memory on the node that executes the pod.

```yaml
apiVersion: v1
kind: LimitRange
metadata:
  name: memory-limit-range
  namespace: ys-namespace
spec:
  limits:
  - default:
      memory: 500Mi
      cpu: 1
    defaultRequest:
      memory: 250Mi
      cpu: 0.4
    type: Container
```

To delete the resourcequota commands
1) kubectl get limitrange -n ys-namespace
2) kubectl delete limitrange memory-limit-range -n ys-namespace

---

12Q) What are the different object familiar

1) Pods
2) Services
3) Replica Set
4) Limit Range
5) Namespace
6) Resource Quota
7) Deployment
8) Config Map
9) Secrets
10) PV
11) PVC
12) Storage Class
13) Stateful Set
14) Headless

13Q) Want to restrict or allow only 6 pods deployed in namespace

need to update in resource quote
hard
limit pods = 2

```yaml
  hard:
    requests.cpu: "1"
    requests.memory: 1Gi
    limits.cpu: "2"
    limits.memory: 2Gi
```

14Q) Limit the containers

Limit range Min max and control

---

15) How do we create a pod in a particular node

Node Selector

1) We need to define a label to node
2) We can define nodeselector in pod spec section in .yaml file so that the pod is created in that particular node
3) If we have more than one node with same label then pod will be created based on the load (scheduler will decide)

Step #1 - First label the node using kubectl command
#kubectl label nodes = #kubectl label nodes node2.example.com size=large
Step #2: Now create a pod definition, specifying the node selector for the pod.

There are limitations

You cannot use the condition that "place the pod on node either "large" or "medium" but not "small". You cannot use multiple checks. For that we use "affinity" and "anti-affinity"

```
spec:
  tolerations:
  - key: "color"
    operator: "Equal"
    value: "red"
    effect: "NoSchedule"
  containers:
```

Steps

1) kubectl get nodes

```
root@ip-10-0-0-31:~# kubectl get nodes
NAME                                        STATUS   ROLES                  AGE    VERSION
ip-10-0-0-122.ap-south-1.compute.internal   Ready    <none>                 4d7h   v1.22.17
ip-10-0-0-31.ap-south-1.compute.internal    Ready    control-plane,master   5d1h   v1.22.17
ip-10-0-0-95.ap-south-1.compute.internal    Ready    <none>                 4d8h   v1.22.17
```

2) kubectl taint node ip-10-0-0-122.ap-south-1.compute.internal colour=red:NoSchedule = **(taint command)**

```
root@ip-10-0-0-31:~# kubectl taint node ip-10-0-0-122.ap-south-1.compute.internal colour=red:NoSchedule
node/ip-10-0-0-122.ap-south-1.compute.internal tainted
```

3) kubectl get pods -n ys-namespace

```
root@ip-10-0-0-31:~# kubectl get pods -n ys-namespace
NAME                                    READY   STATUS    RESTARTS      AGE
tomcat-nginx                            1/1     Running   2 (8h ago)    27h
tomcatdeployment-944dbf64c-fhnmr        1/1     Running   0             71s
tomcatdeployment-944dbf64c-qx4sm        1/1     Running   0             71s
```

4) kubectl get pods -owide -n ys-namespace **= to check which pod is running**

```
root@ip-10-0-0-31:~# kubectl get pods -owide -n ys-namespace
NAME                                    READY   STATUS    RESTARTS      AGE     IP              NODE                                        NOMINATED
 NODE   READINESS GATES
tomcat-nginx                            1/1     Running   2 (8h ago)    27h     10.244.130.205  ip-10-0-0-95.ap-south-1.compute.internal    <none>
       <none>
tomcatdeployment-944dbf64c-fhnmr        1/1     Running   0             2m49s   10.244.130.207  ip-10-0-0-95.ap-south-1.compute.internal    <none>
       <none>
tomcatdeployment-944dbf64c-qx4sm        1/1     Running   0             2m49s   10.244.130.208  ip-10-0-0-95.ap-south-1.compute.internal    <none>
       <none>
```

5) kubectl delete -f deployment.yaml **= To delete the yaml file**

6) kubectl taint node ip-10-0-0-122.ap-south-1.compute.internal colour=red:NoSchedule- **= to taint**

---

16) Affinity

For a variety of reasons a service or container should only run on a specific type of hardware. Maybe one machine has faster disk speeds, another is running a conflicting application, and yet another is part of your bare-metal cluster.

A "Hard rule" – Node Affinity Required during Scheduling Ignored during Execution, which means the rule is "required during scheduling" but has no effect on an already-running Pod. (Hard rule means, it must be met for a pod to be scheduled onto the given node)

A "Soft rule" – Node Affinity Preferred during Scheduling Ignored during Execution, which means the rule is "preferred during scheduling" but likewise has no effect on an already-running pod. (Soft rule means if the scheduler can't satisfy our constraints, the pod will still be schedule to a different node)

Together, these rules are called NodeAffinity because they indicate a Pod's "attraction" to certain Nodes

```
    spec:
      containers:
        - name: tomcat
          image: krishna123456/tomcat:1.11.111
      affinity:
        nodeAffinity:
          preferredDuringSchedulingIgnoredDuringExecution:
            - weight: 1
              preference:
                matchExpressions:
                - key: size
                  operator: NotIn
                  values:
                    - small
                    - large
```

Commands

1) kubectl delete -f deployment.yaml
2) kubectl apply -f deployment.yaml
3) kubectl get pods -owide -n ys-namespace

```
root@ip-10-0-0-31:~# kubectl get pods -owide -n ys-namespace
NAME                                  READY   STATUS    RESTARTS      AGE   IP              NODE                                        NOMINATED
NODE   READINESS GATES
tomcat-nginx                          1/1     Running   2 (11h ago)   30h   10.244.130.205  ip-10-0-0-95.ap-south-1.compute.internal    <none>
       <none>
tomcatdeployment-c766c759c-hwxlc      1/1     Running   0             5s    10.244.130.212  ip-10-0-0-95.ap-south-1.compute.internal    <none>
       <none>
tomcatdeployment-c766c759c-jqztv      1/1     Running   0             5s    10.244.130.211  ip-10-0-0-95.ap-south-1.compute.internal    <none>
       <none>
```

16a) podAffinity

All pods with the same label should be grouped together and deployed

```
spec:
  containers:
    - name: tomcat
      image: krishna123456/tomcat:1.11.111
  affinity:
    podAffinity:
      requiredDuringSchedulingIgnoredDuringExecution:
      - labelSelector:
            matchExpressions:
            - key: app
              operator: In
              values:
              - tomcat
        topologyKey: "kubernetes.io/hostname"
```

Commands

1) kubectl apply -f podAffinity.yaml

2) kubectl get pods -n ys-namespace

16b) podantiAffinity

In Same Pod, with 2 labels should not deployed

```
spec:
  containers:
    - name: tomcat
      image: krishna123456/tomcat:1.11.111
  affinity:
    podAntiAffinity:
      requiredDuringSchedulingIgnoredDuringExecution:
      - labelSelector:
            matchExpressions:
            - key: app
              operator: In
              values:
              - tomcat
        topologyKey: "kubernetes.io/hostname"
```

Commands

1) kubectl apply -f podAffinity.yaml

2) kubectl get pods -n ys-namespace

---

17) Daemonset

1) A daemonset runs a copy of pod in all the nodes of cluster.

2) It ensures that all node run that copy of pod which is created by daemonset.

example: running logs collection daemon/pod on every node running a node monitoring daemon/pod on every node.

```yaml
apiVersion: apps/v1
kind: DaemonSet
metadata:
  name: tomcatdeployment
  namespace: ys-namespace
spec:
  selector:
    matchLabels:
      app: tomcat
  template:
    metadata:
      name: tomcat-nginix
      labels:
        app: tomcat
    spec:
      containers:
        - name: tomcat
          image: ravindra45/ravindra45:1.1.1
```

Commands

1) kubectl apply -f daemonset.yaml

2) kubectl get pods -n ys-namespace

Priority	Class Name	Description
1 (highest)	Guaranteed	If limits and optionally requests are set (not equal to 0) for all resources and they are equal.
2	Burstable	If requests and optionally limits are set (not equal to 0) for all resources, and they are not equal
3 (lowest)	BestEffort	If requests and limits are not set for any of the resources

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: tomcat-nginx
  namespace: ys-namespace
  labels:
    app: tomcat
spec:
  containers:
  - name: mytomcat
    image: prajwal1327/mytomcat:1.1.6
    resources:
      limits:
        memory: "250Mi"
      requests:
        memory: "150Mi"
```

Commands:

1) kubectl apply -f myfirstpod.yaml

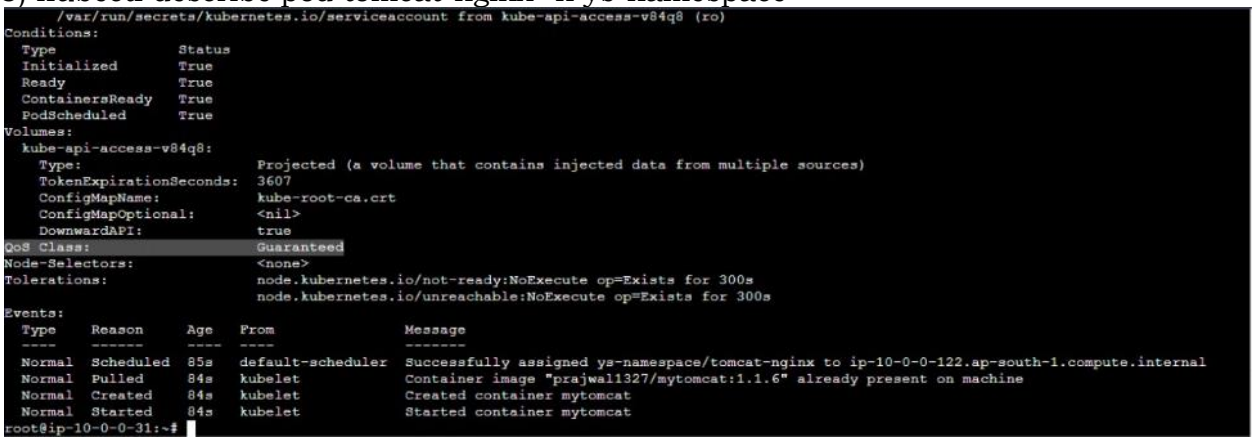
2) kubectl get pod -n ys-namespace
3) kubectl describe pod tomcat-nginx -n ys-namespace

1)

```
resources:
    limits:
      cpu: "400m"
      memory: "250Mi"
    requests:
      memory: "250Mi"
      cpu: "400m"
```
2:47 pm

Commands:

1) kubectl apply -f myfirstpod.yaml
2) kubectl get pod -n ys-namespace
3) kubectl describe pod tomcat-nginx -n ys-namespace



```
      /var/run/secrets/kubernetes.io/serviceaccount from kube-api-access-v84q8 (ro)
Conditions:
  Type              Status
  Initialized       True
  Ready             True
  ContainersReady   True
  PodScheduled      True
Volumes:
  kube-api-access-v84q8:
    Type:                    Projected (a volume that contains injected data from multiple sources)
    TokenExpirationSeconds:  3607
    ConfigMapName:           kube-root-ca.crt
    ConfigMapOptional:       <nil>
    DownwardAPI:             true
QoS Class:                   Guaranteed
Node-Selectors:              <none>
Tolerations:                 node.kubernetes.io/not-ready:NoExecute op=Exists for 300s
                             node.kubernetes.io/unreachable:NoExecute op=Exists for 300s
Events:
  Type    Reason     Age   From               Message
  ----    ------     ----  ----               -------
  Normal  Scheduled  85s   default-scheduler  Successfully assigned ys-namespace/tomcat-nginx to ip-10-0-0-122.ap-south-1.compute.internal
  Normal  Pulled     84s   kubelet            Container image "prajwal1327/mytomcat:1.1.6" already present on machine
  Normal  Created    84s   kubelet            Created container mytomcat
  Normal  Started    84s   kubelet            Started container mytomcat
root@ip-10-0-0-31:~#
```

2)

```
resources:
    limits:
      memory: "250Mi"
    requests:
      memory: "150Mi"
```
2:48 pm

```
      /var/run/secrets/kubernetes.io/serviceaccount from kube-api-access-lxb77 (ro)
Conditions:
  Type              Status
  Initialized       True
  Ready             True
  ContainersReady   True
  PodScheduled      True
Volumes:
  kube-api-access-lxb77:
    Type:                    Projected (a volume that contains injected data from multiple sources)
    TokenExpirationSeconds:  3607
    ConfigMapName:           kube-root-ca.crt
    ConfigMapOptional:       <nil>
    DownwardAPI:             true
QoS Class:                   Burstable
Node-Selectors:              <none>
Tolerations:                 node.kubernetes.io/not-ready:NoExecute op=Exists for 300s
                             node.kubernetes.io/unreachable:NoExecute op=Exists for 300s
Events:
  Type    Reason     Age   From               Message
  ----    ------     ----  ----               -------
  Normal  Scheduled  10s   default-scheduler  Successfully assigned ys-namespace/tomcat-nginx to ip-10-0-0-122.ap-south-1.compute.internal
  Normal  Pulled     10s   kubelet            Container image "prajwal1327/mytomcat:1.1.6" already present on machine
  Normal  Created    10s   kubelet            Created container mytomcat
  Normal  Started    10s   kubelet            Started container mytomcat
root@ip-10-0-0-31:~#
```

Commands:

1) kubectl apply -f myfirstpod.yaml
2) kubectl get pod -n ys-namespace
3) kubectl describe pod tomcat-nginx -n ys-namespace

4) kubectl delete all –all – delete all

---

18) Data Volume

Steps

1) create EFS and attach to the security to the instances (worker Nodes).

A) basic_pod (One Server)

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: jenkins
  labels:
    app: jenkins
spec:
  volumes:
    - name: "jenkins-data"
      hostPath:
        path: "/root/jenkins"
  nodeSelector:
    size: small
  containers:
  - name: jenkins
    image: jenkins/jenkins:lts
    volumeMounts:
      - mountPath: "/var/jenkins_home"
        name: "jenkins-data"
```

Commands

1) kubectl get nodes --show-labels

```
root@ip-10-0-0-31:~# kubectl get nodes --show-labels
NAME                                STATUS   ROLES                  AGE    VERSION    LABELS
ip-10-0-0-122.ap-south-1.compute.internal   Ready    <none>                 6d8h   v1.22.17   beta.kubernetes.io/arch=amd64,beta.kubernetes.io/
instance-type=t2.medium,beta.kubernetes.io/os=linux,failure-domain.beta.kubernetes.io/region=ap-south-1,failure-domain.beta.kubernetes.io/zone=
ap-south-1a,kubernetes.io/arch=amd64,kubernetes.io/hostname=ip-10-0-0-122.ap-south-1.compute.internal,kubernetes.io/os=linux,node.kubernetes.io
/instance-type=t2.medium,size=small,topology.kubernetes.io/region=ap-south-1,topology.kubernetes.io/zone=ap-south-1a
ip-10-0-0-31.ap-south-1.compute.internal    Ready    control-plane,master   7d2h   v1.22.17   beta.kubernetes.io/arch=amd64,beta.kubernetes.io/
os=linux,kubernetes.io/arch=amd64,kubernetes.io/hostname=ip-10-0-0-31.ap-south-1.compute.internal,kubernetes.io/os=linux,node-role.kubernetes.i
o/control-plane=,node-role.kubernetes.io/master=,node.kubernetes.io/exclude-from-external-load-balancers=
ip-10-0-0-95.ap-south-1.compute.internal    Ready    <none>                 6d8h   v1.22.17   beta.kubernetes.io/arch=amd64,beta.kubernetes.io/
instance-type=t2.medium,beta.kubernetes.io/os=linux,failure-domain.beta.kubernetes.io/region=ap-south-1,failure-domain.beta.kubernetes.io/zone=
ap-south-1a,kubernetes.io/arch=amd64,kubernetes.io/hostname=ip-10-0-0-95.ap-south-1.compute.internal,kubernetes.io/os=linux,node.kubernetes.io/
instance-type=t2.medium,topology.kubernetes.io/region=ap-south-1,topology.kubernetes.io/zone=ap-south-1a
```

2) kubectl get nodes

3) kubectl apply -f basicpod.yaml

4) kubectl get pods –owide

```
root@ip-10-0-0-31:~# kubectl get pods -owide
NAME      READY   STATUS    RESTARTS   AGE   IP              NODE                                        NOMINATED NODE   READINESS GATES
jenkins   1/1     Running   0          3s    10.244.146.219   ip-10-0-0-122.ap-south-1.compute.internal   <none>           <none>
root@ip-10-0-0-31:~# kubectl get pods -owide
NAME      READY   STATUS    RESTARTS   AGE   IP              NODE                                        NOMINATED NODE   READINESS GATES
jenkins   1/1     Running   0          7s    10.244.146.219   ip-10-0-0-122.ap-south-1.compute.internal   <none>           <none>
```

```
root@ip-10-0-0-122:~# ls
snap
root@ip-10-0-0-122:~# cd jenkins
root@ip-10-0-0-122:~/jenkins# ls
config.xml                     jenkins.telemetry.Correlator.xml  nodes          secret.key              updates      war
copy_reference_file.log        jobs                              plugins        secret.key.not-so-secret  userContent
hudson.model.UpdateCenter.xml  nodeMonitors.xml                  queue.xml.bak  secrets                 users
root@ip-10-0-0-122:~/jenkins#
```

i-037a197c2c4c4f24e (Worker2)

PublicIPs: 65.2.149.14 PrivateIPs: 10.0.0.122

B) deployment

Disadvantages: All volume should be placed on single node.

```
    matchLabels:
      app: jenkins
  strategy:
    type: RollingUpdate
    rollingUpdate:
      maxSurge: 1
      maxUnavailable: 1
  template:
    metadata:
      name: jenkins
      labels:
        app: jenkins
    spec:
      volumes:
      - name: "jenkins-data"
        hostPath:
          path: "/root/jenkins"
      nodeSelector:
        size: small
      containers:
      - name: jenkins
        image: jenkins/jenkins:lts
        volumeMounts:
        - mountPath: "/var/jenkins_home"
          name: "jenkins-data"
```

Commands:
1) kubectl apply -f deployment1.yaml
2) kubectl get pods -n ys-namespace

```
root@ip-10-0-0-31:~# kubectl get pods -n ys-namespace
NAME                          READY   STATUS    RESTARTS       AGE
jenkins-665648dd94-8g8xx      1/1     Running   0              7s
jenkins-665648dd94-sjddm      1/1     Running   0              7s
tomcat-nginx                  1/1     Running   1 (22h ago)    27h
```

3) kubectl exec -it jenkins-5484866f5d-96bwx -n ys-namespace -- /bin/sh

C) ConfigMap
1) A configmap is an API object used to store non-confidential data in key-value pairs
2) Pods can consume ConfigMaps as Environment Variables, Command-line argument, or as configuration files in a volume.

https://kubernetes.io/docs/concepts/configuration/configmap/

```yaml
apiVersion: v1
kind: ConfigMap
metadata:
  name: game-demo
data:
  # property-like keys; each key maps to a simple value
  player_initial_lives: "3"
  ui_properties_file_name: "user-interface.properties"

  # file-like keys
  game.properties: |
    enemy.types=aliens,monsters
    player.maximum-lives=5
  user-interface.properties: |
    color.good=purple
    color.bad=yellow
    allow.textmode=true
```

Commands
1) kubectl apply -f configmap.yml
2) kubectl get configmap

```
root@ip-10-0-0-31:~# kubectl get configmap
NAME                 DATA   AGE
game-demo            4      30s
kube-root-ca.crt     1      7d3h
```

D) ConfigMaps and Pods

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: configmap-demo-pod
spec:
  containers:
    - name: demo
      image: alpine
      command: ["sleep", "3600"]
      env:
        # Define the environment variable
        - name: PLAYER_INITIAL_LIVES # Notice that the case is different here
                                     # from the key name in the ConfigMap.
          valueFrom:
            configMapKeyRef:
              name: game-demo           # The ConfigMap this value comes from.
              key: player_initial_lives # The key to fetch.
        - name: UI_PROPERTIES_FILE_NAME
          valueFrom:
            configMapKeyRef:
              name: game-demo
              key: ui_properties_file_name
      volumeMounts:
      - name: config
```

```
        mountPath: "/config"
        readOnly: true
  volumes:
  # You set volumes at the Pod level, then mount them into containers inside that Pod
  - name: config
    configMap:
      # Provide the name of the ConfigMap you want to mount.
      name: game-demo
      # An array of keys from the ConfigMap to create as files
      items:
      - key: "game.properties"
        path: "game.properties"
      - key: "user-interface.properties"
        path: "user-interface.properties"
```

Commands

1) kubectl apply -f pod-config.yaml

2) kubectl get pods

```
root@ip-10-0-0-31:~# kubectl get pods
NAME                 READY   STATUS              RESTARTS   AGE
configmap-demo-pod   0/1     ContainerCreating   0          25s
```

3) kubectl get configmap

```
root@ip-10-0-0-31:~# kubectl get configmap
NAME               DATA   AGE
game-demo          4      9s
kube-root-ca.crt   1      7d3h
```

4) kubectl exec -it configmap-demo-pod -- /bin/sh

```
root@ip-10-0-0-31:~# kubectl exec -it configmap-demo-pod -- /bin/sh
/ # ls
bin      dev      home     media    opt      root     sbin     sys      usr
config   etc      lib      mnt      proc     run      srv      tmp      var
```

E) Secrets

1) Secret is an API object used to store confidential data in key-value pairs

2) It is pod specific. No other pod will be able to read the secret value unless and until specified in .yaml explicitly.

secret.yaml
apiVersion:
kind: secret
metadata:

data:

---

19) Kubernetes Persistant Volumes

A PersistentVolume (PV) is a piece of storage in the cluster that has been provisioned by an administrator or dynamically provisioned using Storage Classes.
This API object captures the details of the implementation of the storage, be that NFS, iSCSI, or a cloud-provider-specific storage system.

A PersistentVolumeClaim (PVC) is a request for storage by a user. It is similar to a Pod. Pods consume node resources and PVCs consume PV resources.
Pods can request specific levels of resources (CPU and Memory). Claims can request specific size and access modes

Lifecycle of a volume and claim
PVs are resources in the cluster. PVCs are requests for those resources and also act as claim checks to the resource.
The interaction between PVs and PVCs follows this lifecycle

Provisioning
There are two ways PVs may be provisioned: statically or dynamically.
A) Static
A cluster administrator creates a number of PVs. They carry the details of the real storage, which is available for use by cluster users.
They exist in the Kubernetes API and are available for consumption.

B) Dynamic
When none of the static PVs the administrator created match a user's PersistentVolumeClaim, the cluster may try to dynamically provision a volume specially for the PVC.
This provisioning is based on StorageClasses: the PVC must request a storage class and the administrator must have created and configured that class for dynamic provisioning to occur.

Binding
A user creates, or in the case of dynamic provisioning, has already created, a PersistentVolumeClaim with a specific amount of storage requested and with certain access modes.
A control loop in the master watches for new PVCs, finds a matching PV (if possible), and binds them together.
If a PV was dynamically provisioned for a new PVC, the loop will always bind that PV to the PVC.

Using
Pods use claims as volumes. T
he cluster inspects the claim to find the bound volume and mounts that volume for a Pod.
For volumes that support multiple access modes, the user specifies which mode is desired when using their claim as a volume in a Pod.
---
20) The access modes are

ReadWriteOnce

the volume can be mounted as read-write by a single node. ReadWriteOnce access mode still can allow multiple pods to access the volume when the pods are running on the same node.

ReadOnlyMany

the volume can be mounted as read-only by many nodes.

ReadWriteMany

the volume can be mounted as read-write by many nodes.

ReadWriteOncePod

the volume can be mounted as read-write by a single Pod. Use ReadWriteOncePod access mode if you want to ensure that only one pod across whole cluster can read that PVC or write to it.

This is only supported for CSI volumes and Kubernetes version 1.22+.

---

21) Stateful Applications:

1) Stateful applications saves the user session data at the server side

2) If servers get down it is difficult to transfer the user sessions

3) This type won't work if we want to auto-scale our application servers.

or

StateFulSet:

It is introduced from 1.19 version of k8s

Statefulset is the workload api object used to manage stateful applications

Manages the deployment and scaling of a set of pods and provides guarantees about the ordering and uniqueness of these pods.

Like deployment, a statefulset manages pods that are based on an identical container spec. unlike deployment, a statefulset maintains a sticky identity for each of their pods. These pods are created from the same spec but are not interchangeable. Each has a persistent identifier that it maintains across any rescheduling.

StatefulSets are valuable for applications that require one or more of the following.

Stable, unique network identifiers.

Stable, persistent storage.

Ordered, graceful deployment and scaling.

Ordered, automated rolling updates.

```yaml
apiVersion: v1
kind: Service
metadata:
  name: nginx
  labels:
    app: nginx
spec:
  ports:
  - port: 80
    name: web
  clusterIP: None
  selector:
    app: nginx
```

```yaml
---
apiVersion: apps/v1
kind: StatefulSet
metadata:
  name: web
spec:
  serviceName: "nginx"
  replicas: 2
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: registry.k8s.io/nginx-slim:0.8
        ports:
        - containerPort: 80
          name: web
        volumeMounts:
        - name: www
          mountPath: /usr/share/nginx/html
 volumeClaimTemplates:
 - metadata:
     name: www
   spec:
     accessModes: [ "ReadWriteOnce" ]
     resources:
       requests:
         storage: 1Gi
```

---

22) Stateless Applications:

1) Stateless application does not store any user data at server side.

2) Using a single authentication gateway or client token methodology, data can be stored to validate the users once for multiple microservices.

---

23) Probe

A probe is a diagnostic performed periodically by the kubelet on a container.

Types of probe

The kubelet can optionally perform and react to three kinds of probes on running containers:

1) livenessProbe

Indicates whether the container is running.

If the liveness probe fails, the kubelet kills the container, and the container is subjected to its restart policy.
If a container does not provide a liveness probe, the default state is Success.

2) readinessProbe
Indicates whether the container is ready to respond to requests.
If the readiness probe fails, the endpoints controller removes the Pod's IP address from the endpoints of all Services that match the Pod.
The default state of readiness before the initial delay is Failure. If a container does not provide a readiness probe, the default state is Success.

3) startupProbe
Indicates whether the application within the container is started.
All other probes are disabled if a startup probe is provided, until it succeeds.
If the startup probe fails, the kubelet kills the container, and the container is subjected to its restart policy.
If a container does not provide a startup probe, the default state is Success.

Configure Probes
Probes have a number of fields that you can use to more precisely control the behavior of startup, liveness and readiness checks:

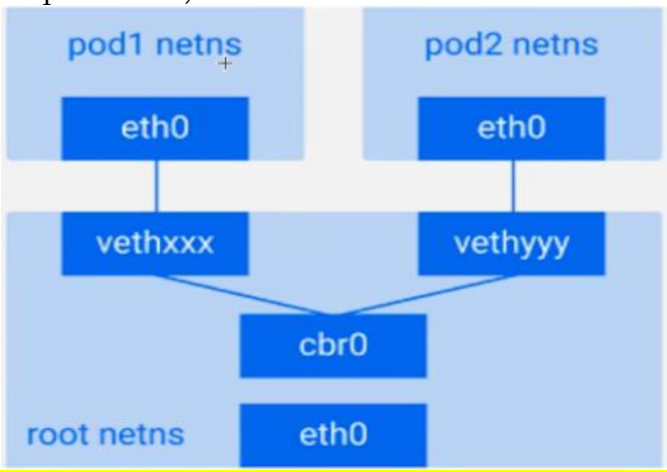
initialDelaySeconds: Number of seconds after the container has started before startup, liveness or readiness probes are initiated. Defaults to 0 seconds. Minimum value is 0.
periodSeconds: How often (in seconds) to perform the probe. Default to 10 seconds. Minimum value is 1.
timeoutSeconds: Number of seconds after which the probe times out. Defaults to 1 second. Minimum value is 1.
successThreshold: Minimum consecutive successes for the probe to be considered successful after having failed. Defaults to 1. Must be 1 for liveness and startup Probes. Minimum value is 1.
failureThreshold: After a probe fails failureThreshold times in a row, Kubernetes considers that the overall check has failed: the container is not ready / healthy / live.

---

24) Kubernetes Networking
There are 5 essential things to understand about networking in Kubernetes

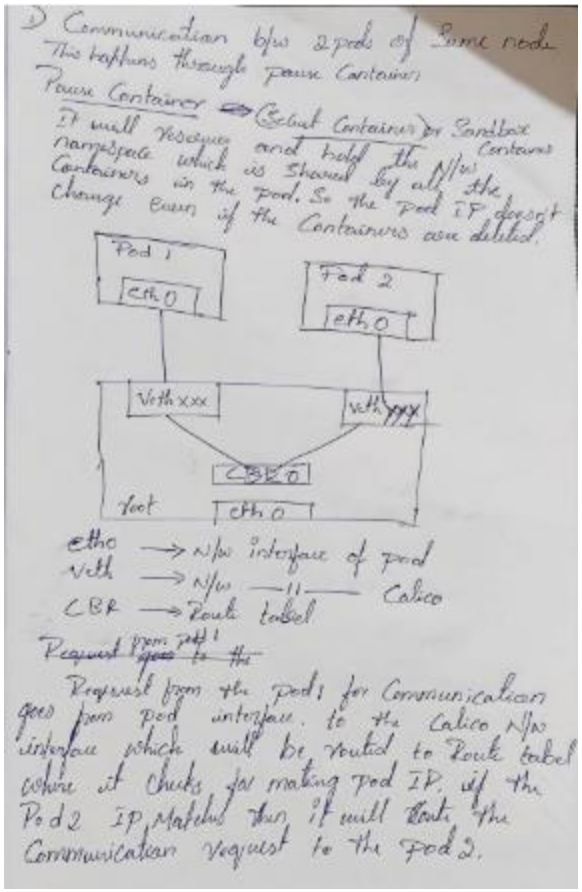
1) Communication between containers on same pod
Answer: Multiple containers in the same Pod share the same IP address. They can communicate with each other by addressing localhost.
For example, if a container in a Pod wants to reach another container in the same Pod

on port 8080, it can use the address localhost:8080



2) Communication between pods on same node

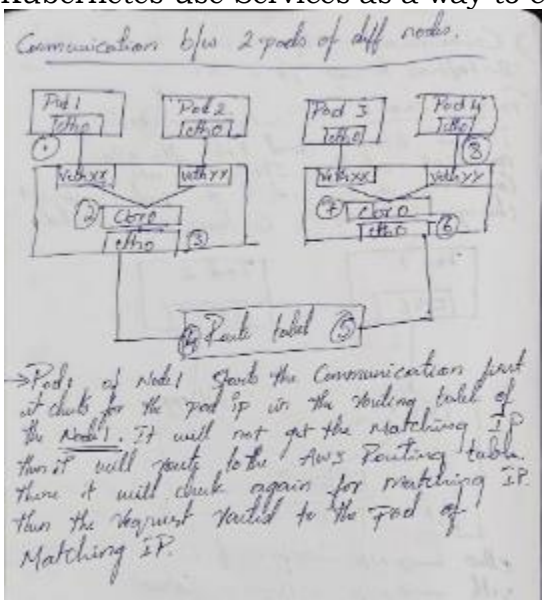
Answer: Pod-to-pod communication within the same node goes through the bridge by default. Let's say we have two pods that have their own network namespaces. When pod 1 wants to talk to pod 2, the packet passes through pod 1's namespace to the corresponding veth pair, vethXXXX , and eventually goes to the bridge.



3) Communication between pods on different nodes

A Pod can communicate with another Pod by directly addressing its IP address, but the recommended way is to use Services. A Service is a set of Pods, which can be reached by a single, fixed DNS name or IP address. In reality, most applications on

Kubernetes use Services as a way to communicate with each other.



4) Communication between pods and services

A Kubernetes service maps a single IP address to a pod group to address the issue of a pod's changing address. The service creates a single endpoint, an immutable IP address and hostname, and forwards requests to a pod in that service.

How does DNS work and how do we discover IP addresses?

Answer: There is a secret container that runs on every pod in k8s.
These containers first job is to keep the namespace open in case all the other containers or pods die. Its called pause containers.
Every pod has a unique IP.
This pod IP is shared by all containers in this pod, and its route table from all the other pods.
Pause containers are called sandbox containers, whose only job is to reserve and hold a network namespace which is shared by all the containers in a pod. This way, a pod IP doesn't change a container dies and a new container is created in a space.
A huge benefits of the IP per pod is there are no IP of port collision with the underlying ports and we don't have to worry about what port the application use.

---

25) What is Ingress

Ingress exposes HTTP and HTTPS routes from outside the cluster to services within the cluster. Traffic routing is controlled by rules defined on the Ingress resource.

An Ingress does not expose arbitrary ports or protocols. Exposing services other than HTTP and HTTPS to the internet typically uses a service of type Service.Type=NodePort or Service.Type=LoadBalancer.

https://raw.githubusercontent.com/kubernetes/ingress-nginx/controller-v1.5.1/deploy/static/provider/aws/deploy.yaml

---

26) Amazon ECR

Amazon Elastic Container Registry (Amazon ECR) is an AWS managed container image

registry service that is secure, scalable, and reliable. Amazon ECR supports private repositories with resource-based permissions using AWS IAM.

END OF Kubernetes

Operating System Volumes: EBS volumes can be used as the root volume for an EC2 instance, allowing you to run your operating system on an EBS volume.

Database Storage: EBS volumes can be used to store data for databases like MySQL, Oracle, or PostgreSQL.

Application Data Storage: EBS volumes can be used to store data that is generated or processed by your applications, such as logs, backups, or backups of databases.

Big Data Storage: EBS volumes can be used to store large amounts of data, such as those generated by big data analytics workloads.

Persistent Storage for Stateful Applications: EBS volumes can be used to store data that must persist even if the EC2 instance is terminated, such as for stateful applications like web servers, applications servers, and caching servers.

Backup and Recovery: EBS volumes can be used as the target for backup and recovery operations, such as snapshots and backups of databases.

Dev/Test Environments: EBS volumes can be used for dev/test environments, where you can create snapshots of your production data and use them to create test and development environments.

---

policies are embedded directly within a single IAM user, group, or role. Inline policies are tightly scoped to the entity they are associated with, and are useful for simple permission scenarios where only a single policy is needed.

Here's an example of an inline policy:

Suppose you have an S3 bucket named "my-bucket" and you want to grant an IAM user named "jane" the permission to list the objects in the bucket and download them. You can create the following inline policy for the IAM user "jane":

json
Copy code

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "s3:ListBucket",
                "s3:GetObject"
            ],
```

```
            "Resource": [
                "arn:aws:s3:::my-bucket",
                "arn:aws:s3:::my-bucket/*"
            ]
        }
    ]
}
```

This policy allows the IAM user "jane" to list the objects in the "my-bucket" S3 bucket and download the objects. The policy uses the "Effect" element to specify the action (allow) and the "Action" element to specify the allowed actions (s3:ListBucket and s3:GetObject). The "Resource" element specifies the S3 bucket and objects that the policy applies to.

Create a hosted zone: A hosted zone in Route 53 is a container for DNS records for a specific domain. To handle traffic management for multiple domains or subdomains, you would create a separate hosted zone for each domain or subdomain you want to manage.

Create DNS records: For each hosted zone, you would create one or more DNS records that map domain names to IP addresses or other resources. For example, you could create an "A" record that maps a subdomain, such as "www.example.com", to an IP address.

Route traffic to resources: Using Route 53, you can route traffic to your resources, such as EC2 instances or S3 buckets, based on the DNS records you have created. For example, you could create a weighted round robin routing policy to distribute traffic evenly across multiple resources for a subdomain.

Monitor and adjust traffic: Route 53 provides health checking and traffic flow features to help you monitor the health of your resources and adjust traffic as needed. For example, you can configure Route 53 to stop routing traffic to an unhealthy resource and automatically route traffic to a healthy resource instead.

Create a health check: To monitor the health of a resource, you create a health check in Route 53. A health check specifies the endpoint (such as a web server or an application) to monitor and the parameters for checking the endpoint's health. For example, you might create a health check that monitors an HTTP endpoint and checks for a 200 OK response code.

Check the endpoint's health: Route 53 periodically performs health checks on the endpoint you specified. If the endpoint is healthy, it returns a status of "healthy" and Route 53 continues to route traffic to the endpoint. If the endpoint is not healthy, it returns a status of "unhealthy" and Route 53 stops routing traffic to the endpoint.

Route traffic based on health: Route 53 uses the health check status to determine how to route traffic to your resources. If the endpoint is healthy, Route 53 routes traffic to the endpoint. If the endpoint is unhealthy, Route 53 stops routing traffic to the endpoint and can route traffic to another healthy endpoint instead.

Monitor the health of your resources: Route 53 provides you with visibility into the health of your resources, so you can monitor the health of your resources and take action if needed. You can view the results of health checks and the health of your resources in the Route 53 console or programmatically.

EC2: You can use Route 53 to route traffic to your Amazon Elastic Compute Cloud (EC2) instances. For example, you can create an "A" record that maps a domain name to an EC2 instance's IP address, and then use Route 53 to route traffic to that EC2 instance. You can also use Route 53 health checks to monitor the health of your EC2 instances and route traffic to healthy instances.

S3: You can use Route 53 to route traffic to your Amazon Simple Storage Service (S3) buckets. For example, you can create an "A" record that maps a subdomain to an S3 bucket, and then use Route 53 to route traffic to that S3 bucket. You can also use Route 53 health checks to monitor the health of your S3 buckets and route traffic to healthy buckets.

ELB: You can use Route 53 to route traffic to your Amazon Elastic Load Balancer (ELB). For example, you can create an "A" record that maps a domain name to an ELB endpoint, and then use Route 53 to route traffic to the ELB. Route 53 automatically routes traffic to the healthy instances behind the ELB, and monitors the health of the ELB endpoint using Route 53 health checks.

Private hosted zones in Amazon Route 53 allow you to create and manage custom DNS domains within your Amazon Virtual Private Cloud (VPC). The domains are not publicly resolvable, but can be used to route traffic within your VPC.

Here's when you should use private hosted zones:

Isolation of internal resources: If you have internal resources within your VPC that should not be accessible from the Internet, you can use a private hosted zone to route traffic to those resources. The private hosted zone provides isolation for your internal resources, so that only instances within your VPC can resolve the domain names associated with those resources.

Route traffic within a VPC: You can use private hosted zones to route traffic between instances within a VPC, without having to use public IP addresses. This makes it easier to manage and route traffic within your VPC.

Multiple VPCs: If you have multiple VPCs, you can use private hosted zones to route traffic between VPCs, without having to traverse the Internet. This can be useful for creating hybrid cloud environments, where some resources are located in one VPC and other resources are located in another VPC.

Amazon Route 53 Resolver is a managed DNS service that provides domain name resolution for resources within and between Amazon Virtual Private Clouds (VPCs) and on-premises networks connected to AWS.

Route 53 Resolver works by providing a scalable and secure way to resolve domain names between VPCs and on-premises networks. It uses a network of resolver endpoints to resolve domain names to IP addresses.

Here's how it works:

A request for domain name resolution is sent to a resolver endpoint in a VPC or on-premises network.

The resolver endpoint forwards the request to the Route 53 Resolver service.

The Route 53 Resolver service checks its internal DNS database for the IP address associated with the requested domain name.

If the IP address is found, the Route 53 Resolver service returns it to the resolver endpoint.

The resolver endpoint then returns the IP address to the requesting application or system.

Route 53 Resolver provides a scalable, secure, and managed way to resolve domain names between VPCs and on-premises networks connected to AWS, helping to ensure that applications and systems have fast and reliable access to the resources they need.

Global Network: Route 53 uses a global network of DNS servers to resolve domain names to IP addresses. This ensures that DNS queries are handled quickly and efficiently, regardless of the location of the requesting client.

Anycast: Route 53 uses an anycast network, which routes incoming DNS queries to the nearest available DNS server. This minimizes the time it takes to resolve a query, reducing latency and improving response times.

Health Checking: Route 53 monitors the health of resources and automatically routes traffic away from unhealthy resources to healthy ones. This ensures that DNS queries are resolved to healthy resources, even if one or more resources become unavailable.

Load Balancing: Route 53 can distribute incoming DNS queries across multiple resources, improving the overall performance and reliability of the system.

Failover: Route 53 can detect when a resource becomes unavailable and automatically route traffic to an alternate resource, ensuring that DNS queries continue to be resolved even if one or more resources are unavailable.

Domain Name System (DNS) Spoofing Attacks: To protect against DNS spoofing attacks, it is recommended to enable DNSSEC.

DDoS attacks: To protect against DDoS attacks, it is recommended to use Amazon Route 53's traffic flow features and/or to use Amazon CloudFront as a content delivery network (CDN).

Data privacy: It is important to keep in mind that information about the domains and DNS records managed by Route 53 are stored in the AWS infrastructure, and as such, are subject to Amazon's privacy policies.

Access control: It is recommended to use IAM policies and AWS resource-level permissions to control access to Route 53 resources and API actions.

Secure Transfer of Data: When transferring DNS data to Route 53,

Create a hosted zone in Route 53 for your domain: You can create a new hosted zone in Route 53 for your domain and configure the name servers for your domain to delegate to the Route 53 name servers.

Export your current DNS records: You can use the export functionality of your current DNS provider to get a file that contains your current DNS records.

Import your current DNS records: In Route 53, you can import your current DNS records into the hosted zone that you created in step 1.

Update the name server records for your domain: Once you have imported your current DNS records into Route 53, you need to update the name server records for your domain to delegate to the Route 53 name servers. This step is performed at your domain registrar.

Monitor the migration: After you have updated the name server records for your domain, it can take up to 48 hours for the changes to propagate globally. You should monitor the migration process to ensure that it is successful.

Create a VPC: Create a VPC with multiple subnets in different availability zones.

Public Subnet: Create a public subnet for the front-end tier. In this subnet, launch EC2 instances that host your web application and make them reachable from the Internet. You may also want to place a load balancer in this subnet to distribute incoming traffic.

Private Subnet: Create a private subnet for the back-end tier. In this subnet, launch EC2 instances that host your back-end services and make them not reachable from the Internet.

Database Subnet: Create a separate subnet for the database tier. In this subnet, launch a database instance, such as Amazon RDS, and make it only accessible from the private subnet.

Network Access Control: Configure network access control, such as security groups and network ACLs, to control the traffic flow between the subnets and to the Internet. For example, you can allow incoming traffic from the Internet to reach the public subnet, but block incoming traffic from the Internet to the private subnet.

Internet Gateway: Attach an Internet Gateway to the VPC to allow communication between the public subnet and the Internet.

Routing: Configure routing between the subnets using route tables and set the appropriate routes for traffic flow between the subnets and to the Internet.

Create a VPC: Create a VPC with multiple subnets in different availability zones.

Dev/Test Subnets: Create separate subnets for dev and test environments. In each subnet, launch EC2 instances that host your dev/test applications and configure network access control, such as security groups and network ACLs, to limit access to only the necessary resources.

Production Subnets: Create separate subnets for production environment. In each subnet, launch EC2 instances that host your production applications and configure network access control, such as security groups and network ACLs, to limit access to only the necessary resources.

outline for connecting an on-premise network to an Amazon Virtual Private Cloud (VPC) using Direct Connect and VPN Connections:

Direct Connect: To connect your on-premise network to an Amazon VPC using Direct Connect, you need to create a Direct Connect connection. You'll need to order a Direct Connect circuit from a Direct Connect partner and create a virtual interface to connect to your VPC. This provides a dedicated, low-latency and high-bandwidth network connection between your on-premise network and Amazon VPC.

VPN Connections: To connect your on-premise network to an Amazon VPC using VPN Connections, you need to create a Virtual Private Network (VPN) connection. This can be achieved by creating a virtual private gateway and a customer gateway in the VPC and on-premise network, respectively. Then, create a VPN connection to link the virtual private gateway and customer gateway, and configure routing to allow communication between your on-premise network and Amazon VPC.

Routing: After you have established the connection between your on-premise network and Amazon VPC, you need to configure routing to allow communication between the two networks. This can be done by updating the route tables in the VPC and on-premise network to reflect the new connection.

Create a VPC endpoint: Start by creating a VPC endpoint for the AWS service you want to access. This creates a network connection between your VPC and the AWS service, allowing you to access the service without having to go over the internet.

Configure security groups: Next, configure security groups to control access to the VPC endpoint. This can be used to limit access to specific IP addresses or to specific instances within your VPC.

Route network traffic: Finally, update your VPC routing tables to route network traffic to the VPC endpoint. This allows instances within your VPC to access the AWS service through the VPC endpoint.

Create an S3 bucket: Create a new Amazon S3 bucket with a unique and meaningful name, and configure it to host a static website by setting the bucket's property "Static Website Hosting" to "Enabled."

Upload the website: Upload the website files, such as HTML, CSS, and JavaScript files, to the newly created S3 bucket. You can use the AWS Management Console or the AWS CLI to upload the files.

Configure the custom domain: To use a custom domain for your website, you need to configure the domain name in Amazon Route 53. Create a new hosted zone for the domain name and add an alias record that points to the S3 bucket.

Enable HTTPS encryption: To enable HTTPS encryption for your website, you need to obtain a security certificate for the custom domain. You can obtain a certificate from Amazon Certificate Manager (ACM) or a third-party certificate authority. After obtaining the certificate, you need to configure the S3 bucket to use the certificate.

Update the S3 bucket policy: To ensure that only authorized users can access the S3 bucket, you need to update the S3 bucket policy. For example, you can specify that only users with the necessary AWS IAM permissions can access the bucket and its content.

Create source and target S3 buckets: First, create two Amazon S3 buckets, one in the primary region (source) and one in a secondary region (target). The source bucket should contain the critical business data that needs to be protected.

Enable cross-region replication: Next, enable cross-region replication for the source bucket by setting the bucket's replication configuration. In the replication configuration, specify the target bucket, the AWS identity that will be used to perform the replication, and the desired replication rule. The replication rule defines the criteria for what objects will be replicated, such as all objects in the bucket or objects with a specific prefix.

Configure versioning: To ensure that all versions of the objects are preserved during replication, enable versioning for both the source and target buckets.

Monitor replication status: After the replication is set up, monitor the replication status regularly to ensure that the critical business data is being replicated as expected. You can use Amazon CloudWatch or Amazon S3 event notifications to monitor the replication status.

Test disaster recovery: To validate the disaster recovery solution, you can periodically run disaster recovery tests by making changes to the source bucket and verifying that the changes are replicated to the target bucket.

Create an AWS Organizations structure: To manage multiple AWS accounts, you can use AWS Organizations to centralize account management and to simplify the sharing of resources across accounts.

Create a centralized IAM policy: In the master account of the AWS Organizations, create an IAM policy that defines the permissions for accessing the S3 buckets. The policy should specify the S3 actions that are allowed and the resources (S3 buckets) that the policy applies to.

Create a custom IAM role: In each member account, create a custom IAM role that allows the users in that account to assume the role and access the S3 buckets in the master account. The role should include a trust relationship that allows the users in the member account to assume the role.

Assign the policy to the role: Attach the centralized IAM policy created in step 2 to the custom IAM role created in step 3. This will allow the users in each member account to access the S3 buckets in the master account.

Grant permissions to users: In each member account, grant permissions to the users who need access to the S3 buckets by creating IAM users and attaching the custom IAM role created in step 3 to the users.

```
{
   "Version": "2012-10-17",
   "Statement": [
      {
         "Effect": "Allow",
         "Action": "s3:GetObject",
         "Resource": "arn:aws:s3:::example-bucket/*",
         "Condition": {
            "StringLike": {
               "aws:SourceIp": "192.0.2.0/24"
            },
            "StringEquals": {
               "aws:UserAgent": "AWS Console"
            }
         }
      }
   ]
}
```

Define conditions for geographic location and IP address: In the IAM policy, you can use the "Condition" element to define the geographic location and IP address conditions for accessing the AWS resources. For example, you can specify that the policy is applicable only if the request originates from a specific IP address range or a specific geographic region.

Create a policy statement: In the policy statement, you can specify the AWS actions and resources that the policy applies to, and also the conditions that must be met in order for the policy to be enforced. For example, you can specify that the policy allows only the "GetObject" action for S3 buckets, and only if the request originates from a specific IP address range or geographic region.

Attach the policy to a user, group, or role: After you have created the policy, you can attach it to a user, group, or role. This will allow the users in the group, or the entities that assume the role, to access the AWS resources only if the conditions defined in the policy are met.
