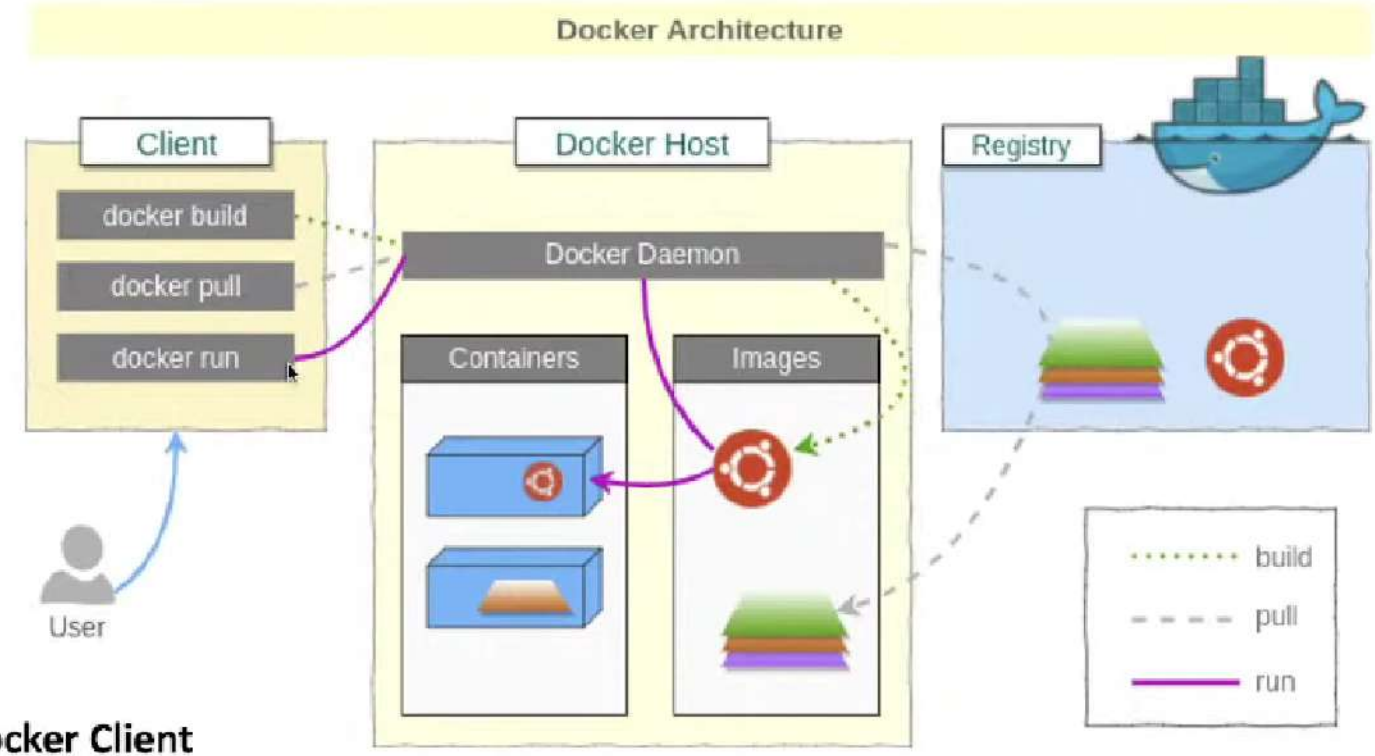
Docker Architecture

- Docker uses a client-server architecture.
- The Docker *client* talks to the Docker *daemon*, which does the heavy lifting of building, running, and distributing your Docker containers. The Docker client and daemon *can* run on the same system, or you can connect a Docker client to a remote Docker daemon.
- The Docker client and daemon communicate using a REST API, over UNIX sockets or a network interface. Another Docker client is Docker Compose, that lets you work with applications consisting of a set of containers.



1) **Docker Client**
2) **Docker Daemon**
3) **Docker Registry**
4) **Docker Image**

Docker Image/Container and DockerFile Commands

- A Docker image is a read-only, inert template that comes with instructions for deploying containers. In Docker, everything basically revolves around images.
- An image consists of a collection of files (or layers) that pack together all the necessities—such as dependencies, source code, and libraries—needed to set up a completely functional container environment.
- Images are stored on a Docker registry, such as the Docker Hub, or on a local registry.

```
PS C:\WINDOWS\system32> docker pull bash:5.0
5.0: Pulling from library/bash
188c0c94c7c5: Pulling fs layer
94387ca39817: Pulling fs layer
efe7174943e6: Pulling fs layer
efe7174943e6: Verifying Checksum
efe7174943e6: Download complete
188c0c94c7c5: Verifying Checksum
188c0c94c7c5: Download complete
188c0c94c7c5: Pull complete
94387ca39817: Verifying Checksum
94387ca39817: Download complete
94387ca39817: Pull complete
efe7174943e6: Pull complete
Digest: sha256:01fad26fa8ba21bce6e8c47222acfdb54649957f1e88d53a0c8e03360271abf6
Status: Downloaded newer image for bash:5.0
docker.io/library/bash:5.0
```

```
PS C:\WINDOWS\system32> docker images
REPOSITORY          TAG                 IMAGE ID
bash                5.0                 39a95ac32011

PS C:\WINDOWS\system32>
```

A Docker container is a virtualized runtime environment that provides isolation capabilities for separating the execution of applications from the underpinning system. It's an instance of a Docker image.

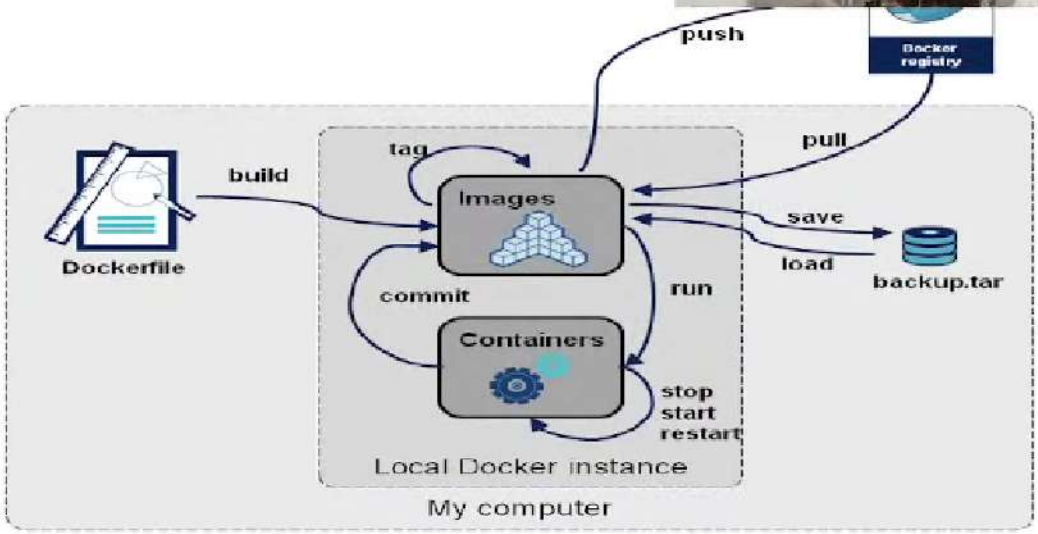
```
Base Image      Image Version Pinning missing (DL3006,DL3007)
                FROM ubuntu :12.04

                Maintainer or maintainer email missing (DL3012,D4000)
                MAINTAINER John Doe <joe@doe.org>
Env. Variable   ENV USE_HTTP 0

Comment         # Add proxy settings
                COPY  ./setenv.sh /tmp/

`RUN` can execute any shell command
                RUN sudo apt-get update
                RUN sudo apt-get upgrade -y
Installing dependencies
                RUN apt-get install -y wget :1.12
                  Dependency Version Pinning missing (DL3008,DL3013)
                RUN sudo -E pip install scipy :0.18.1
Installing software (compiling, linking, etc.)
                RUN cd /usr/src/vertica-sqlalchemy;
                    sudo python ./setup.py install

Open Port       EXPOSE 8888
                # CMD ipython notebook --ip=* …
                ADD runcmd.sh /   Using ADD instead of COPY (DL3020)
                RUN chmod u+x /runcmd.sh
```

Docker Main Co

- **docker pull** ubuntu. (To pull image from hub/repo)
- **docker run** –it –name c1 –d –p 82:80 ubuntu (To run an image as container)
- docker exec –it c1 bash (To login into container)

  Backup of container as Image

- **docker commit** c1 apache-on-ubuntu:1.0(To save the container data as new Image)
- **docker save** apache-on-ubuntu:1.0 --output backup.tar (To save the image as tar)
- **docker load** -i backup.tar (To unzip the image from tar)
- **docker start/stop/restart** c1 (Conatiner commands)
- **docker push** image-name (To push the image to conatiner)
- **docker build** Dockerfile



Docker Container States

- **Created -** Docker assigns the *created* state to the containers that were never started ever since they were created. Hence, no CPU or memory is used by the containers in this state.
- **Running -** When we start a container having created a state using the docker start command, it attains the running state. This state signifies that the processes are running in the isolated environment inside the container.
- **Restarting -** Docker supports four types of restart policies, namely – no, on-failure, always, unless-stopped. Restart policy decides the behaviour of the container when it exit. By default, the restart policy is set to no, which means that the container will not be started automatically after it exits.
- **Exited** - This state is achieved when the process inside the container terminates. **No CPU and memory are consumed** by the container.

The process inside the container was completed, and so it exited.

The process inside the container encountered an exception while running.

A container is intentionally stopped using the docker stop command.

No interactive terminal was set to a container running bash.

- **Pause** - A paused container consumes the same memory used while running the container, but the CPU is released completely

```
$ docker create --name mycontainer httpd
8d60cb560afc1397d6732672b2b4af16a08bf6289a5a0b6b5125c5635e8ee749
$ docker inspect -f '{{.State.Status}}' mycontainer
created
$ docker start mycontainer
mycontainer
$ docker inspect -f '{{.State.Status}}' mycontainer
running
```

```
$ docker run -itd --restart=always --name mycontainer centos:7 sleep 5
f7d0e8becdac1ebf7aae25be2d02409f0f211fcc191aea000041d158f89be6f6
```

```
$ docker pause mycontainer
mycontainer
$ docker inspect -f '{{.State.Status}}' mycontainer
paused
```

```
$ docker stats --no-stream
CONTAINER ID   NAME          CPU %     MEM USAGE / LIMIT   MEM %     NET I/O       BLOCK I/O   PIDS
1a44702cea17   mycontainer   0.00%     1.09MiB / 7.281GiB   0.01%     1.37kB / 0B   0B / 0B     1
```

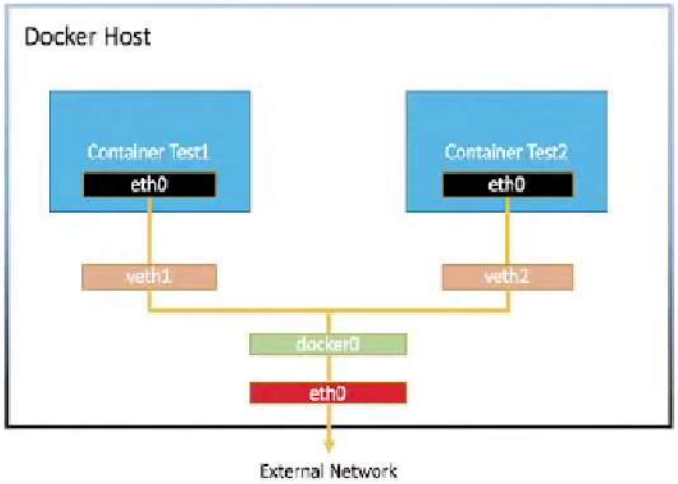
Docker Networking

1) **Bridge**: The default network driver. Bridge networks apply to containers running on the same Docker daemon host

2) **Host**: For standalone containers, remove network isolation between the container and the Docker host, and use the host's networking directly

3) **Overlay**: Overlay networks connect multiple Docker daemons together and enable swarm services to communicate with each other. You can also use overlay networks to facilitate communication between a swarm service and a standalone container, or between two standalone containers on different Docker daemons.

- **User-defined bridges** provide better isolation
- Containers can be attached and detached from user-defined networks on the fly.
- Each user-defined network creates a configurable bridge.



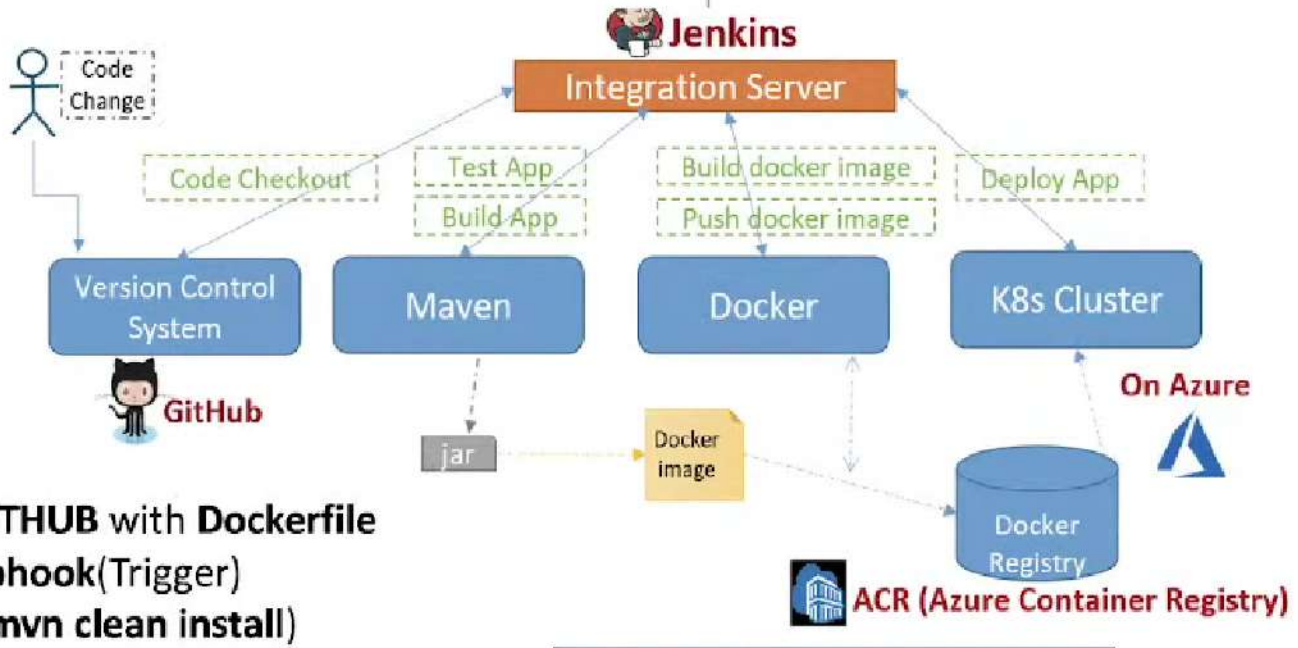
Managing User defined bridge network

- docker network create -d **docker0** (Basic command)
- docker network rm my-net (To remove the network)
- docker network create -d overlay **docker0** (To connect to multiple daemons)
- docker network create --driver=bridge --subnet=172.28.0.0/16 --ip-range=172.28.5.0/24 --gateway=172.28.5.254 **docker0**
- docker create --name my-nginx --network **docker0** --publish 8080:80 nginx:latest (To create a container without start state)
- docker network disconnect **docker0** my-nginx

DockerFile

SpringBootSwagger | SpringBootSwagger | SpringBootSwagger | SpringBootSwagger | Dockerfile

```
1 FROM openjdk:8-jdk-alpine
2 RUN addgroup -S spring && adduser -S spring -G spring
3 USER spring:spring
4 ARG JAR_FILE=target/*.jar
5 COPY ${JAR_FILE} app.jar
6 ENTRYPOINT ["java","-jar","/app.jar"]
```



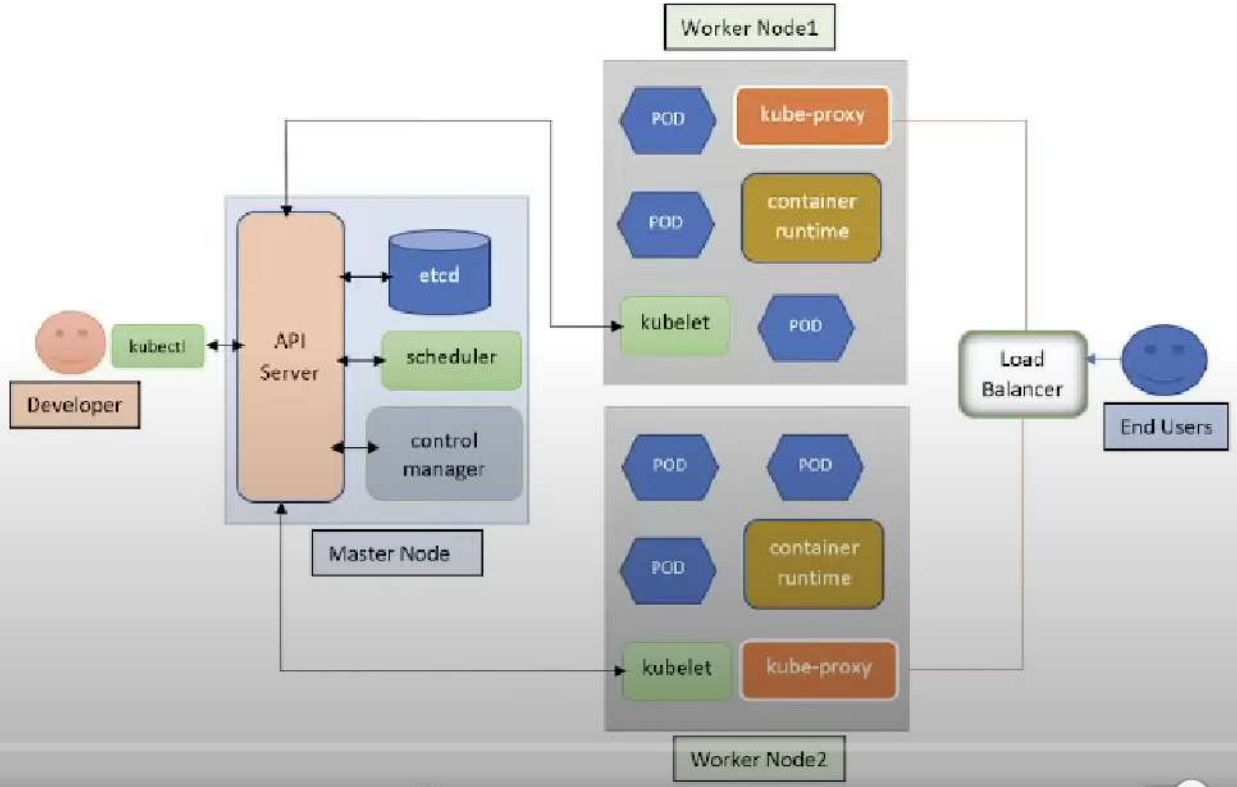
Stage 1 - Developer commits the JAVA code in **GITHUB** with **Dockerfile**
Stage 2 - **Jenkins** will check for changes with **webhook**(Trigger)
Stage 3 - **Maven** build happens with command (**mvn clean install**)
Stage 4 - **Docker build** command is given in Jenkins (docker build .)
  Stage 4.1 -The **docker image** is created with the help of dockerfile
  Stage 4.2 - **Docker push to dockerregistry or Jfrog**
Stage 5(Deploy App) - **ANSIBLE** (Which will pull the image from dockerregistry

SESSION 4- KUBERNETES | FREE DEVOPS BOOTCAMP 😍 | This FREE bootcamp will be 🔥 |Liv...

Press Esc to exit full screen

Kubernetes Architecture:

A Kubernetes cluster consists of a set of worker machines, called nodes, that run containerized applications. Every cluster has at least one worker node.
The worker node(s) host the Pods that are the components of the application workload.
The control pane manages the worker nodes and the Pods in the cluster.



10:26 / 2:10:13

Components of Kubernetes:

- **Master Node Components:**

1. API Server 2. Controller Manager 3. ETCD 4. Scheduler

1) API Server:

It is the front-end for the Kubernetes control plane.

2) Controller Manager: This is a component on the master that runs controllers.

- **Node Controller**: Responsible for noticing and responding when nodes go down.
- **Replication Controller**: Responsible for maintaining the correct number of pods for every replication controller object in the system.
- **Endpoints Controller**: Populates the Endpoints object (that is, it joins Services and Pods).
- **Service Account and Token Controllers:** Create default accounts and API access tokens for new namespaces.

3) **ETCD**: Consistent and highly-available key value store used as Kubernetes' backing store for all cluster data.

4) **kube-scheduler -** Control plane component that watches for newly created Pods with no assigned node, and selects a node for them to run on.

Node Components

Node components run on every node, maintaining running pods and providing the Kubernetes runtime environment

1) **Kubelet** - An agent that runs on each node in the cluster. It makes sure that containers are running in a Pod.

The kubelet takes a set of PodSpecs that are provided through various mechanisms and ensures that the containers described in those PodSpecs are running and healthy. The kubelet doesn't manage containers which were not created by Kubernetes.

2)**kube-proxy** - kube-proxy is a network proxy that runs on each node in your cluster, implementing part of the Kubernetes Service concept. Kube-proxy maintains network rules on nodes. These network rules allow network communication to your Pods from network sessions inside or outside of your cluster.

3)**Container runtime** - The container runtime is the software that is responsible for running containers.

Manifest File Components

apiversion

Kind

Metadata

Spec

File name : nginx-deployment.yaml

```yaml
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: my-nginx
  labels:
    app: nginx
spec:
  replicas: 3
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx:1.14.2
        ports:
        - containerPort: 80
```

Service components:

Kubernetes ServiceTypes allow you to specify what kind of Service you want. The default is ClusterIP.

Type values and their behaviors are:

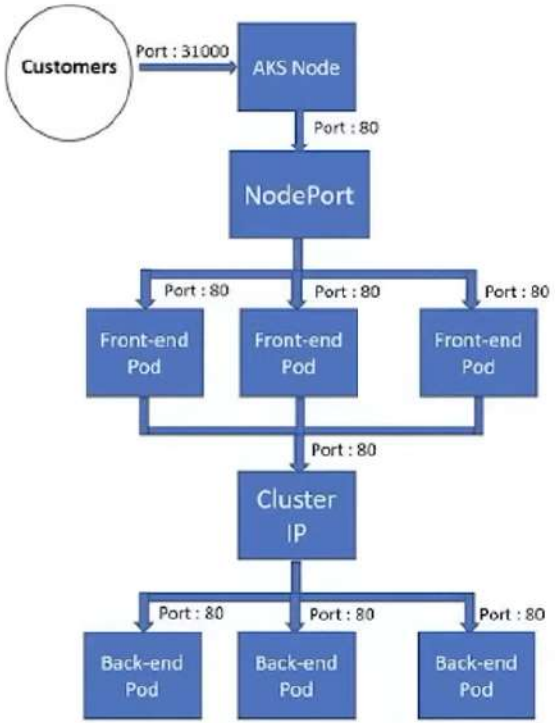
ClusterIP: Exposes the Service on a cluster-internal IP. Choosing this value makes the Service only reachable from within the cluster. This is the default ServiceType.(To talk to other nodes in the cluster)

NodePort: Exposes the Service on each Node's IP at a static port (the NodePort). A ClusterIP Service, to which the NodePort Service routes, is automatically created. You'll be able to contact the NodePort Service, from outside the cluster, by requesting <NodeIP>:<NodePort>. (The entrpoint for node)

```yaml
apiVersion: v1
kind: Service
metadata:
  name: my-service
spec:
  type: NodePort
  selector:
    app: MyApp
  ports:
      # By default and for convenience,
    - port: 80
      targetPort: 80
      # Optional field
      # By default and for convenience,
      nodePort: 30007
```

- **LoadBalancer**: Exposes the Service externally using a cloud provider's load balancer. NodePort and ClusterIP Services, to which the external load balancer routes, are automatically created.
- **Externalname**: Maps the Service to the contents of the externalName field (e.g. foo.bar.example.com), by returning a CNAME record with its value. No proxying of any kind is set up.

```yaml
apiVersion: v1
kind: Service
metadata:
  name: my-nginx-svc
  labels:
    app: nginx
spec:
  type: LoadBalancer
  ports:
  - port: 80
  selector:
    app: nginx
```




```yaml
kind: Service
apiVersion: v1

metadata:
  name: hostname-service

spec:
  type: NodePort
  selector:
    app: echo-hostname
  ports:
    - nodePort: 30163
      port: 8080
      targetPort: 80
```

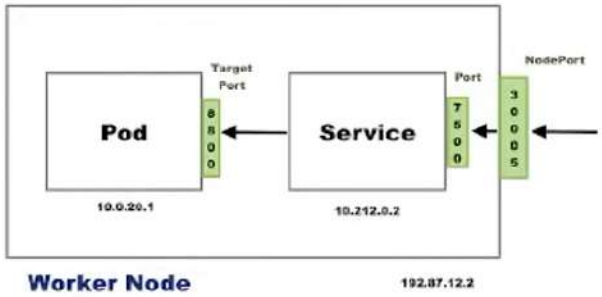
Make the service available to network requests from external clients

Forward requests to pods with label of this value

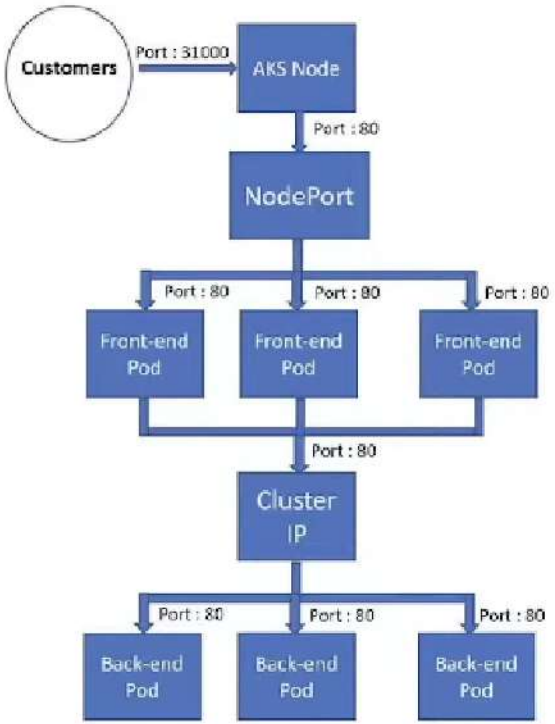
nodePort
access service via this external port number

port
port number exposed internally in cluster

targetPort
port that containers are listening on



Routing In kubernetes




```yaml
kind: Service
apiVersion: v1

metadata:
  name: hostname-service

spec:
  type: NodePort
  selector:
    app: echo-hostname
  ports:
    - nodePort: 30163
      port: 8080
      targetPort: 80
```

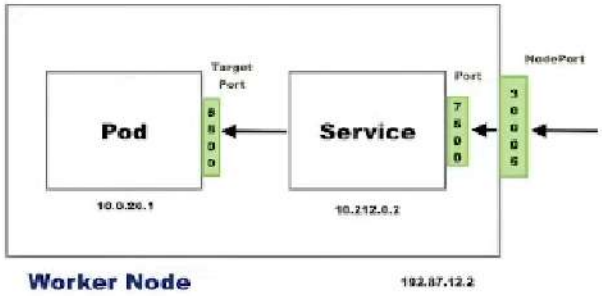
Make the service available to network requests from external clients

Forward requests to pods with label of this value

nodePort
access service via this external port number

port
port number exposed internally in cluster

targetPort
port the containers are listening on



Routing In kubernetes

Kubernetes Commands

1) Kubectl create deploy my-nginx –image=nginx –dry-run=client –o yaml > myngínx.yaml
2) Kubectl get pods
3) Kubectl get namespace
4) Kubectl create ns mydev
5) Kubectl create –f pod.yaml
6) Kubectl describe svc nginx
7) Kubectl exec -it pod1 bash

Kubernetes IQ:

1) What is the architecture of kubernetes
2) What does control manager, etcd, scheduler, API server do
3) What is a manifest file and what are the components of it
4) What is node affinity, pod afiinity , taint toleration
5) What is node port, cluster ip
6) What is persitant volumes and why we use it
7) Describe what is pod and what is pod lifecycle
8) What are the components on master and worked node
9) What is ingress controller
10) What are types of services in kuberntes
11) How one pod talks with other pod
12) How the pod healthcheck is done(describe rediness, livesness)
13) How the monitoring is done(integration on Prometheus and grafana)
14) What is deamonset, replicaset, horizontal pod autoscaler
15) Write a manifest file of your own choice
16) What is namespace and why we use it
17) What are helm charts and uses