**Week-9 Lab Assessment**

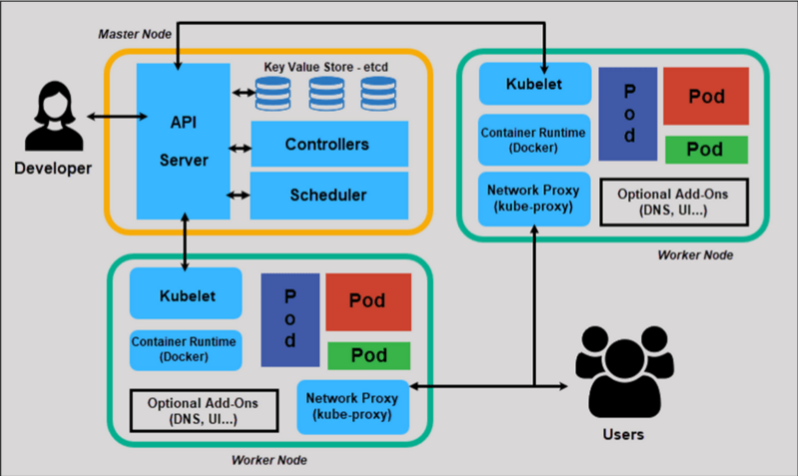
**Assignment-1 Marks: 25**

**Explain the architecture of Kubernetes along with its components and architectural diagram.**

Kubernetes is an open source orchestration tool developed by Google for managing microservices or containerized applications across a distributed cluster of nodes. Kubernetes provides highly resilient infrastructure with zero downtime deployment capabilities, automatic rollback, scaling, and self-healing of containers (which consists of auto-placement, auto-restart, auto-replication , and scaling of containers on the basis of CPU usage).

The main objective of Kubernetes is to hide the complexity of managing a fleet of containers by providing REST APIs for the required functionalities. Kubernetes is portable in nature, meaning it can run on various public or private cloud platforms such as AWS, Azure, OpenStack, or Apache Mesos. It can also run on bare metal machines.

**Kubernetes Components and Architecture :**



Kubernetes follows a client-server architecture. It’s possible to have a multi-master setup (for high availability), but by default there is a single master server which acts as a controlling node and point of contact. The master server consists of various components including a kube-apiserver, an etcd storage, a kube-controller-manager, a cloud-controller-manager, a kube-scheduler, and a DNS server for Kubernetes services. Node components include kubelet and kube-proxy on top of [Docker](https://wiki.aquasec.com/display/containers/Docker+Containers). High level Kubernetes architecture diagram showing a cluster with a master and two worker nodes

**Master components:**

Below are the main components found on the master node:

* [**etcd cluster**](https://kubernetes.io/docs/tasks/administer-cluster/configure-upgrade-etcd/) – a simple, distributed key value storage which is used to store the Kubernetes cluster data (such as number of pods, their state, namespace, etc), API objects and service discovery details. It is only accessible from the API server for security reasons. etcd enables notifications to the cluster about configuration changes with the help of watchers. Notifications are API requests on each etcd cluster node to trigger the update of information in the node’s storage.
* [**kube-apiserver**](https://kubernetes.io/docs/reference/generated/kube-apiserver/) – Kubernetes API server is the central management entity that receives all REST requests for modifications (to pods, services, replication sets/controllers and others), serving as frontend to the cluster. Also, this is the only component that communicates with the etcd cluster, making sure data is stored in etcd and is in agreement with the service details of the deployed pods.
* [**kube-controller-manager**](https://kubernetes.io/docs/reference/generated/kube-controller-manager/) – runs a number of distinct controller processes in the background (for example, replication controller controls number of replicas in a pod, endpoints controller populates endpoint objects like services and pods, and others) to regulate the shared state of the cluster and perform routine tasks. When a change in a service configuration occurs (for example, replacing the image from which the pods are running, or changing parameters in the configuration yaml file), the controller spots the change and starts working towards the new desired state.
* [**cloud-controller-manager**](https://kubernetes.io/docs/concepts/overview/components/#cloud-controller-manager) – is responsible for managing controller processes with dependencies on the underlying cloud provider (if applicable). For example, when a controller needs to check if a node was terminated or set up routes, load balancers or volumes in the cloud infrastructure, all that is handled by the cloud-controller-manager.
* [**kube-scheduler**](https://kubernetes.io/docs/reference/generated/kube-scheduler/) – helps schedule the pods (a co-located group of containers inside which our application processes are running) on the various nodes based on resource utilization. It reads the service’s operational requirements and schedules it on the best fit node. For example, if the application needs 1GB of memory and 2 CPU cores, then the pods for that application will be scheduled on a node with at least those resources. The scheduler runs each time there is a need to schedule pods. The scheduler must know the total resources available as well as resources allocated to existing workloads on each node.

**Node (worker) components**

Below are the main components found on a (worker) node:

* [**kubelet**](https://kubernetes.io/docs/reference/generated/kubelet/) – the main service on a node, regularly taking in new or modified pod specifications (primarily through the kube-apiserver) and ensuring that pods and their containers are healthy and running in the desired state. This component also reports to the master on the health of the host where it is running.
* [**kube-proxy**](https://kubernetes.io/docs/reference/generated/kube-proxy/) – a proxy service that runs on each worker node to deal with individual host subnetting and expose services to the external world. It performs request forwarding to the correct pods/containers across the various isolated networks in a cluster.

**Kubectl**

[**kubectl**](https://kubernetes.io/docs/reference/kubectl/overview/) command is a line tool that interacts with kube-apiserver and send commands to the master node. Each command is converted into an API call.

**Kubernetes Concepts**

Making use of Kubernetes requires understanding the different abstractions it uses to represent the state of the system, such as services, pods, volumes, namespaces, and deployments.

* [**Pod**](https://kubernetes.io/docs/concepts/workloads/pods/pod-overview/) – generally refers to one or more containers that should be controlled as a single application. A pod encapsulates application containers, storage resources, a unique network ID and other configuration on how to run the containers.
* [**Service**](https://kubernetes.io/docs/concepts/services-networking/service/) – pods are volatile, that is Kubernetes does not guarantee a given physical pod will be kept alive (for instance, the replication controller might kill and start a new set of pods). Instead, a service represents a logical set of pods and acts as a gateway, allowing (client) pods to send requests to the service without needing to keep track of which physical pods actually make up the service.
* [**Volume**](https://kubernetes.io/docs/concepts/storage/volumes/) – similar to a container volume in Docker, but a Kubernetes volume applies to a whole pod and is mounted on all containers in the pod. Kubernetes guarantees data is preserved across container restarts. The volume will be removed only when the pod gets destroyed. Also, a pod can have multiple volumes (possibly of different types) associated.
* [**Namespace**](https://kubernetes.io/docs/concepts/overview/working-with-objects/namespaces/) – a virtual cluster (a single physical cluster can run multiple virtual ones) intended for environments with many users spread across multiple teams or projects, for isolation of concerns. Resources inside a namespace must be unique and cannot access resources in a different namespace. Also, a namespace can be allocated a [resource quota](https://kubernetes.io/docs/concepts/policy/resource-quotas/) to avoid consuming more than its share of the physical cluster’s overall resources.
* [**Deployment**](https://kubernetes.io/docs/concepts/workloads/controllers/deployment/) – describes the desired state of a pod or a replica set, in a yaml file. The deployment controller then gradually updates the environment (for example, creating or deleting replicas) until the current state matches the desired state specified in the deployment file. For example, if the yaml file defines 2 replicas for a pod but only one is currently running, an extra one will get created. Note that replicas managed via a deployment should not be manipulated directly, only via new deployments.

**Assignment-2 Marks: 25**

**Build a docker image using dockerfile containing some web application (apart from nginx) and push it to docker hub.**

**Using Jenkins, take that image from docker hub and run it to access at web browser.**

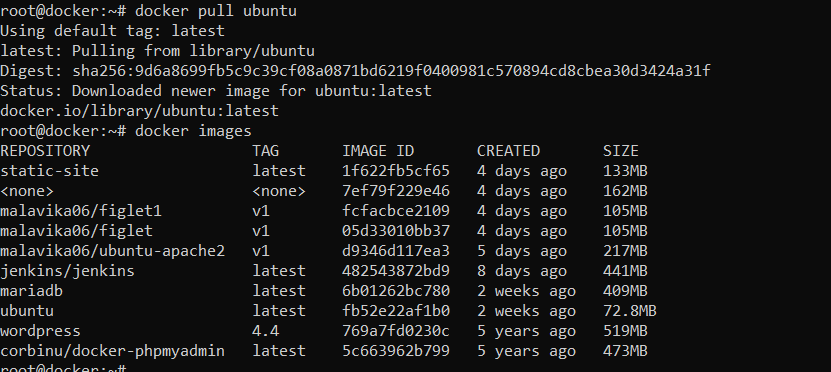
**Assignment-3 Marks: 25**

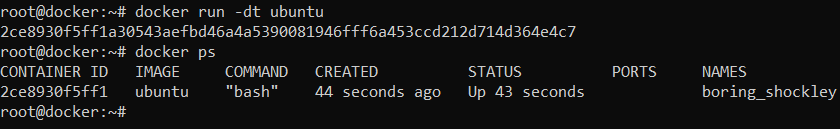
**Create a custom image using ubuntu base image by flowing below steps.**

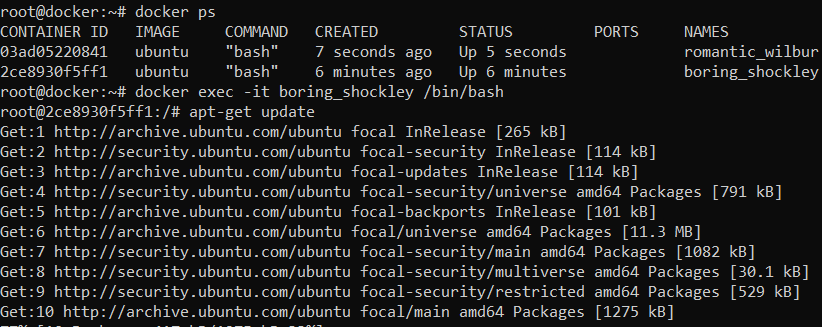
**1. Pull ubuntu image from docker hub and install some package and add a text file “puneet.txt” with the contents “Welcome to DevOps assessment-9” in it.**

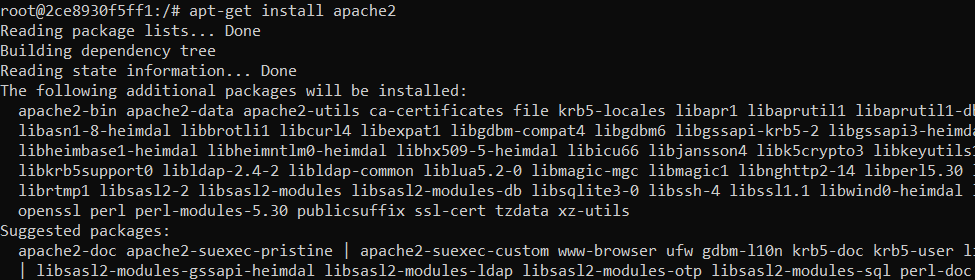
**2. Push it back to the docker hub to create a custom image with the name in the format *<your-docker-id>/ubuntu-wk9-assessment*”.**

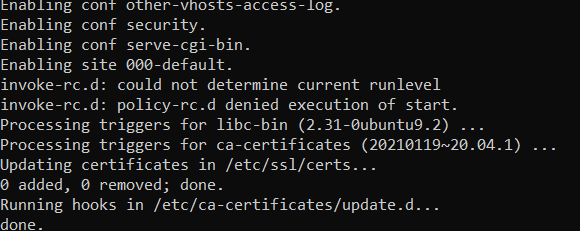
**3. Clean up your local repository and pull this custom image from docker hub and verify the changes made to this custom image.**

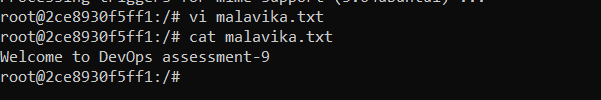


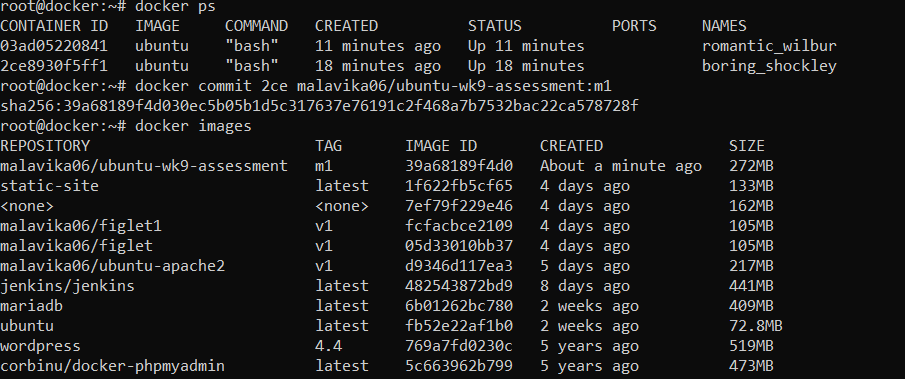


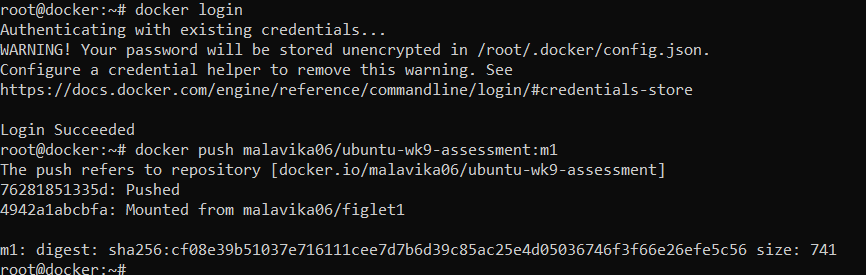


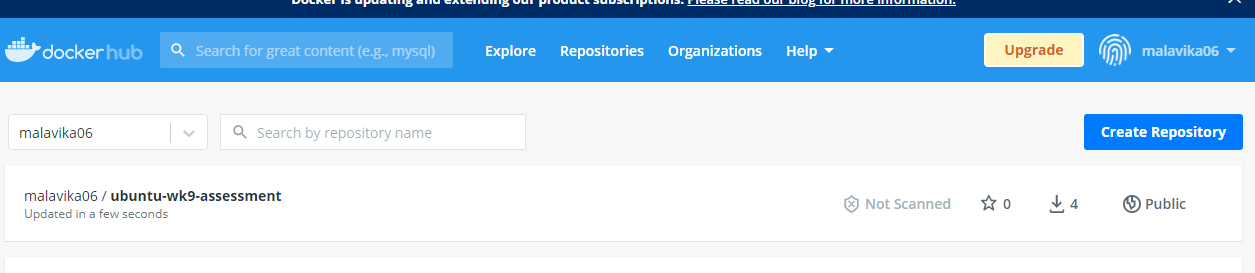


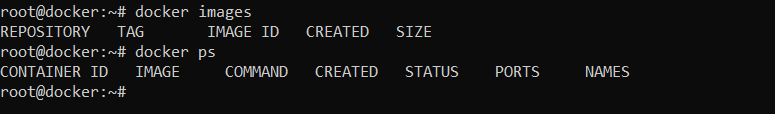












**Assignment-4 Marks: 25**

**Explain the concept of Ingress and its advantages in Kubernetes.**

**Also, demonstrate its use by deploying some application in Kubernetes.**

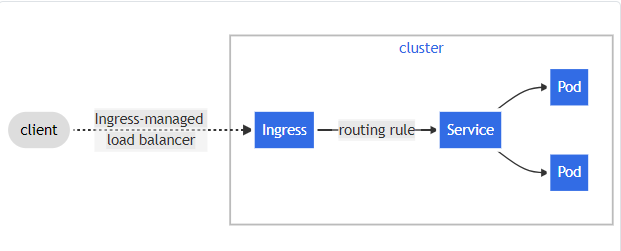
Ingress routes and manages traffic from the outside world to workloads running on Kubernetes

A minimal Kubernetes cluster provides several abstractions for letting applications receive requests from other apps (i.e., inside the cluster) and from the outside world. When developers want to expose an application to traffic (e.g., for testing), they typically define one of these basic services as a starting point:

* ***ClusterIP*** – Assigns a port within a known range. Lets a workload receive requests from other applications and entities inside the cluster.
* ***NodePort***– Lets a workload receive requests from the outside world on a specific port, exposed on all (or a subset of) cluster node IP addresses.
* ***LoadBalancer*** – Assigns an external load balancer (and optionally, DNS name) to the workload’s NodePort. The load balancer, which must be provided by surrounding infrastructure marshaled by a specific infrastructure provider running on the cluster (for example, a Kubernetes cluster running on AWS might integrate with Elastic Load Balancer — another AWS service, via an AWS provider), can then balance requests across nodes exposing the workload.

These primitive service types, however, don’t support all features production applications need. They can’t terminate SSL connections. They can’t support rewrite rules. They can’t (at least not easily) support conditional routing schemes — for example, sending requests to myapp/dothis to one workload, and requests to myapp/dothat to another.

In conventional web environments, features like these are provided by the webserver/proxy (e.g., nginx), by adaptations (e.g., certificates) on the host supporting the webserver, by helper functions like .htaccess, by front-end proxies, etc., or even by applications themselves.

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. Here is a simple example where an Ingress sends all its traffic to one Service:

An Ingress may be configured to give Services externally-reachable URLs, load balance traffic, terminate SSL / TLS, and offer name-based virtual hosting. An [Ingress controller](https://kubernetes.io/docs/concepts/services-networking/ingress-controllers) is responsible for fulfilling the Ingress, usually with a load balancer, though it may also configure your edge router or additional frontends to help handle the traffic.

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](https://kubernetes.io/docs/concepts/services-networking/service/" \l "nodeport) or [Service.Type=LoadBalancer](https://kubernetes.io/docs/concepts/services-networking/service/" \l "loadbalancer).

## Prerequisites

You must have an [Ingress controller](https://kubernetes.io/docs/concepts/services-networking/ingress-controllers) to satisfy an Ingress. Only creating an Ingress resource has no effect.

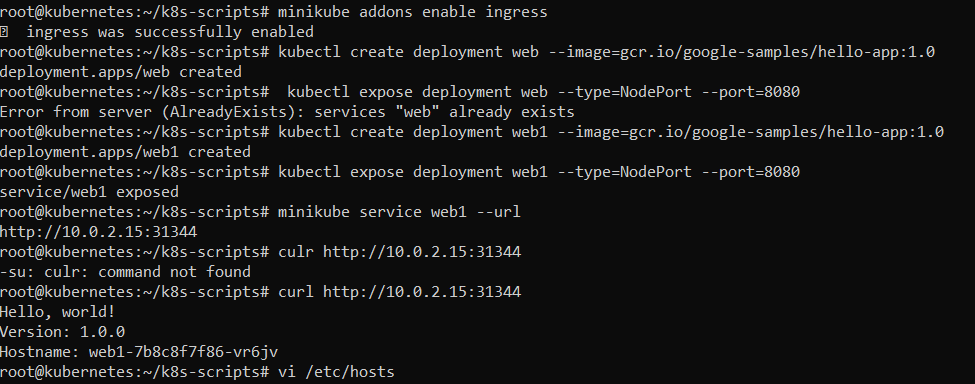
You may need to deploy an Ingress controller such as [ingress-nginx](https://kubernetes.github.io/ingress-nginx/deploy/). You can choose from a number of [Ingress controllers](https://kubernetes.io/docs/concepts/services-networking/ingress-controllers).

Ideally, all Ingress controllers should fit the reference specification. In reality, the various Ingress controllers operate slightly differently.

**Enterprise Kubernetes ingress**

Ingress is a Kubernetes service type designed to solve these problems. It provides a standard way of describing routing, termination, URL-rewriting and other rules in a YAML configuration file, plus standards for building applications/services to read and implement these configurations.

Kubernetes itself doesn’t implement an Ingress solution — most simple cluster models (k0s being a good example) don’t support it, unless users choose and run an ingress controller on their cluster, and integrate with it (nginX is a frequent default choice).

Enterprise Kubernetes solutions are more frequently provided with an ingress controller pre-integrated. An enterprise-grade ingress controller often provides features in excess of those defined by Kubernetes standard ingress, and will provide ways for configuring these additional features alongside basic features, in otherwise-standard ingress configuration files. For example, the well-known Istio ingress controller provides means for blacklisting IP address ranges (perhaps because these IP addresses are recognized as a source of denial-of-service attacks).

