Kubernetes

Capgemini

Course Information

Course Name : Kubernetes

Program Overview: In this program, you will learn the basics of Kubernetes and how to use Kubernetes (k8s) for automating deployment, scaling, and management of containerized applications.

Program Duration: 2 days.

Prerequisite Skills: Minimum 2 years' experience of software development or IT operation. Knowledge of Linux and Windows basic commands. Knowledge of Docker is must.

Who should attend? People interested in automation and meet the above prerequisites.

On Completion of this program: Participants will gain knowledge of Kubernetes and apply it for automating deployment, scaling, and management of containerized applications.

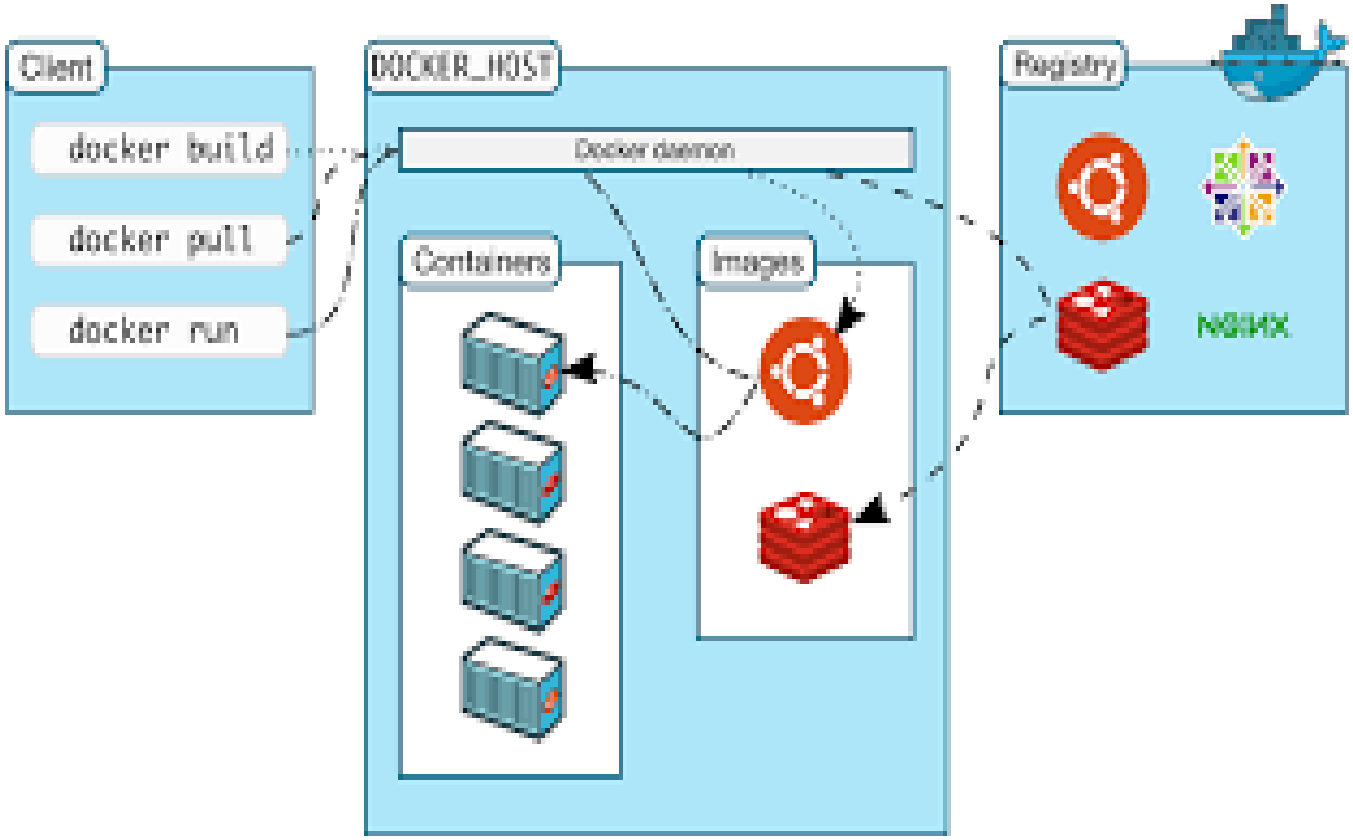
Course Information

- Table of contents:
  - Introduction to Kubernetes
  - Kubernetes architecture and components
  - Kubernetes Installation and Configuration
    - Create a Kubernetes simple environment with Minikube
    - Creating multinode cluster on Linux
    - Cluster management- the Kubectl cli and API structure
  - Accessing the K8S Cluster using the API
  - Kubernetes Web UI – Kubernetes dashboard
  - Working with Kubernetes Objects
    - Pods
    - Deployments
    - Services & Service types
    - Replica Sets
    - Replication Controllers and Deployment
  - Deploying a node js containerized application in K8s cluster
  - Accessing shell of running container
  - Service discovery
  - Kubernetes networking
  - Volumes and Application Data
  - Logging
  - Health Checks

What is docker?

- Docker is a tool designed to make it easier to create, deploy, and run applications by using containers.
- Containers allow a developer to package up an application with all of the parts it needs, such as libraries and other dependencies, and ship it all out as one package.
- By doing so the developer can rest assured that the application will run on any other Linux machine regardless of any customized settings that machine might have that could differ from the machine used for writing and testing the code.
- And importantly, Docker is open source.
- Docker is a tool that is designed to benefit both developers and system administrators, making it a part of many DevOps (developers + operations) toolchains. For developers, it means that they can focus on writing code without worrying about the system. For operations staff, Docker gives flexibility and potentially reduces the number of systems needed because of its small footprint and lower overhead.

Docker Architecture

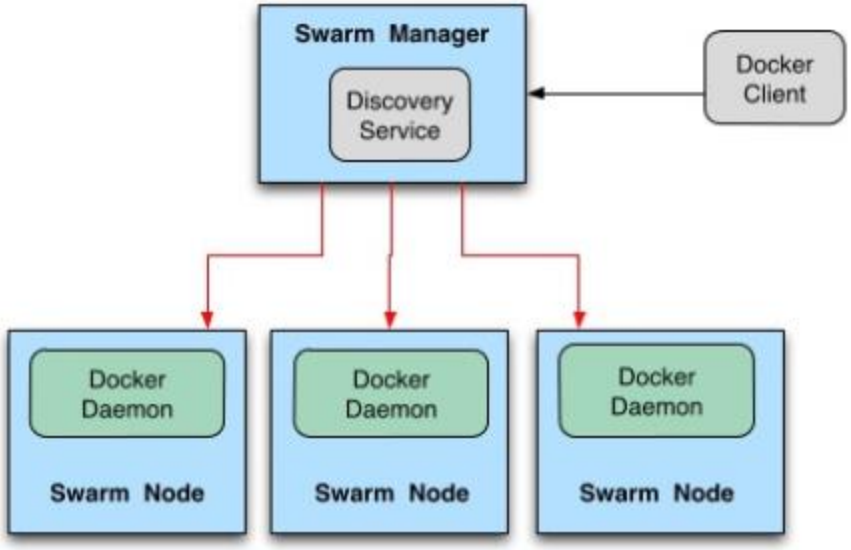


What is Docker Swarm?

- **Docker Swarm**
  - As a platform, Docker has revolutionized the manner software was packaged. Docker Swarm or simply Swarm is an open-source container orchestration platform and is the native clustering engine for and by Docker. Any software, services, or tools that run with Docker containers run equally well in Swarm. Also, Swarm utilizes the same command line from Docker.

  - Swarm turns a pool of Docker hosts into a virtual, single host. Swarm is especially useful for people who are trying to get comfortable with an orchestrated environment or who need to adhere to a simple deployment technique but also have more just one cloud environment or one particular platform to run this on.

Docker Swarm Architecture



Introduction to Kubernetes

What is Kubernetes

- Kubernetes is a powerful 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 v1.0 was released on July 21, 2015. Along with the Kubernetes v1.0 release, Google partnered with the Linux Foundation to form the Cloud Native Computing Foundation (CNCF)

Docker SWARM and Kubernetes

Features	Kubernetes	Docker Swarm
Installation	Complex Installation but a strong resultant cluster once set up	Simple installation but the resultant cluster is not comparatively strong
GUI	Comes with an inbuilt Dashboard	There is no Dashboard which makes management complex
Scalability	Highly scalable service that can scale with the requirements. 5000 node clusters with 150,000 pods	Very high scalability. Up to 5 times more scalable than Kubernetes. 1000 node clusters with 30,000 containers
Load Balancing	Manual load balancing is often needed to balance traffic between different containers in different pods	Capability to execute auto load balancing of traffic between containers in the same cluster
Rollbacks	Automatic rollbacks with the ability to deploy rolling updates	Automatic rollback facility available only in Docker 17.04 and higher if a service update fails to deploy
Logging and Monitoring	Inbuilt tools available for logging and monitoring	Lack of inbuilt tools. Needs 3rd party tools for the purpose
Node Support	Supports up to 5000 nodes	Supports 2000+ nodes
Optimization Target	Optimized for one single large cluster	Optimized for multiple smaller clusters
Updates	The in-place cluster updates have been constantly maturing	Cluster can be upgraded in place
Networking	An overlay network is used which lets pods communicate across multiple nodes	The Docker Daemons is connected by overlay networks and the overlay network driver is used
Availability	High availability. Health checks are performed directly on the pods	High availability. Containers are restarted on a new host if a host failure is encountered

Introduction to Kubernetes

- What can Kubernetes do for you?
  - With modern web services, users expect applications to be available 24/7, and developers expect to deploy new versions of those applications several times a day.
  - Containerization helps package software to serve these goals, enabling applications to be released and updated in an easy and fast way without downtime.
  - Kubernetes helps you make sure those containerized applications run where and when you want, and helps them find the resources and tools they need to work.
  - Kubernetes is a production-ready, open source platform designed with Google's accumulated experience in container orchestration, combined with best-of-breed ideas from the community.

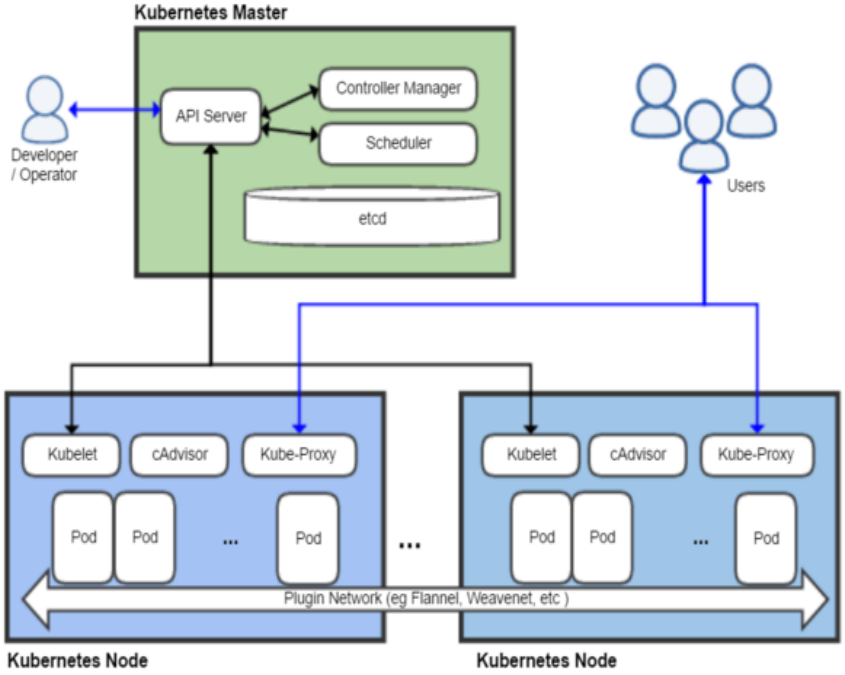
Introduction to Kubernetes

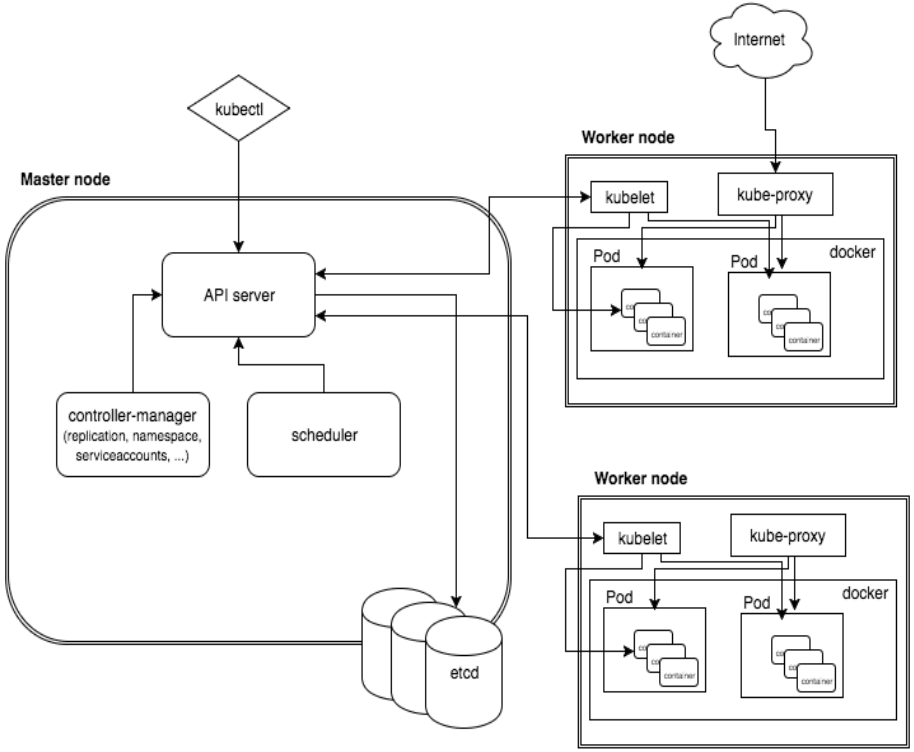
Kubernetes Design Principles

- Kubernetes was designed to support the features required by highly available distributed systems, such as (auto-)scaling, high availability, security and portability.
  - Scalability - Kubernetes provides horizontal scaling of pods on the basis of CPU utilization. When there are multiple pods for a particular application, Kubernetes provides the load balancing capacity across them.
  - High Availability - Kubernetes addresses highly availability both at application and infrastructure level. Replica sets ensure that the desired (minimum) number of replicas of a stateless pod for a given application are running.
  - Security - Kubernetes addresses security at multiple levels: cluster, application and network. The API endpoints are secured through transport layer security (TLS). Only authenticated users (either service accounts or regular users) can execute operations on the cluster (via API requests).
  - Portability - Kubernetes portability manifests in terms of operating system choices (a cluster can run on any mainstream Linux distribution), processor architectures (either virtual machines or bare metal), cloud providers (AWS, Azure or Google Cloud Platform), and new container runtimes, besides Docker, can also be added.

Kubernetes architecture and components

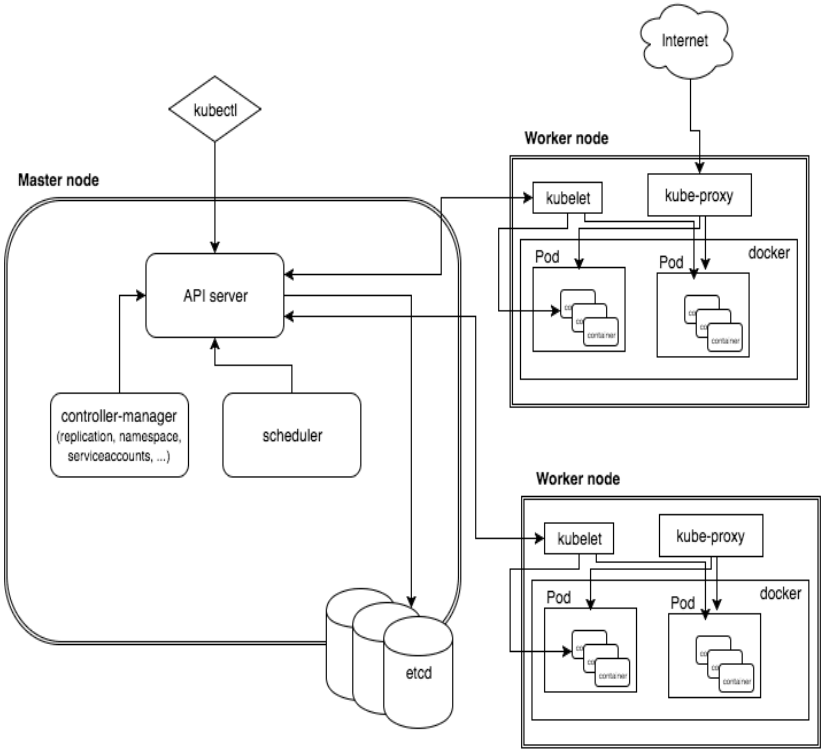
- Kubernetes Architecture





Kubernetes architecture and components

- Kubernetes follows the master-slave architecture. The components of Kubernetes can be divided into those that manage an individual node and those that are part of the control plane.
- Kubernetes control plane (master)
  - The Kubernetes Master is the main controlling unit of the cluster, managing its workload and directing communication across the system. The Kubernetes control plane can run a single master node or on multiple masters supporting high-availability clusters.



Kubernetes architecture and components

- etcd
  - etcd is a persistent, lightweight, distributed, key-value data store that reliably stores the configuration data of the cluster, representing the overall state of the cluster at any given point of time.
  - The Kubernetes API Server uses etcd's watch API to monitor the cluster and roll out critical configuration changes or simply restore any divergences of the state of the cluster, back to what was declared by the deployer.
  - As an example, if the deployer specified that three instances of a particular pod need to be running, this fact is stored in etcd. If it is found that only two instances are running, this delta will be detected by comparison with etcd data, and Kubernetes will use this to schedule the creation of an additional instance of that pod.

Kubernetes architecture and components

- API server:
  - The API server is a key component and serves the Kubernetes API using JSON over HTTP, which provides both the internal and external interface to Kubernetes.
  - The API server processes and validates REST requests and updates state of the API objects in etcd, thereby allowing clients to configure workloads and containers across Worker nodes.
  - 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.

- **Kubectl**
  - **kubectl** is a command line tool that interacts with kube-apiserver and send commands to the master node. Each command is converted into an API call.

Kubernetes architecture and components

- Scheduler
  - The scheduler is the pluggable component that selects which node an unscheduled pod (the basic entity managed by the scheduler) runs on, based on resource availability.
  - Scheduler tracks resource use on each node to ensure that workload is not scheduled in excess of available resources.
  - For this purpose, the scheduler must know the resource requirements, resource availability, and other user-provided constraints and policy directives such as quality-of-service, affinity/anti-affinity requirements, data locality, and so on.
  - In essence, the scheduler's role is to match resource "supply" to workload "demand".

Kubernetes architecture and components

- Controller manager
  - A controller is a reconciliation loop that drives actual cluster state toward the desired cluster state.It does this by managing a set of controllers.
  - One kind of controller is a replication controller, which handles replication and scaling by running a specified number of copies of a pod across the cluster.
  - It also handles creating replacement pods if the underlying node fails.
  - Other controllers that are part of the core Kubernetes system include a "DaemonSet Controller" for running exactly one pod on every machine (or some subset of machines), and a "Job Controller" for running pods that run to completion, e.g. as part of a batch.
  - The controller manager is a process that runs core Kubernetes controllers like DaemonSet Controller and Replication Controller. The controllers communicate with the API server to create, update, and delete the resources they manage (pods, service endpoints, etc.).

Kubernetes architecture and components

- Kubernetes node
  - The Node, also known as Worker or Minion, is a machine where containers (workloads) are deployed.
  - Every node in the cluster must run a container runtime such as Docker, as well as the below-mentioned components, for communication with master for network configuration of these containers.
    - Kubelet
    - Kube-proxy
    - Container runtime

Kubernetes architecture and components

- Kubernetes node
  - Kubelet:
    - Kubelet is responsible for the running state of each node, ensuring that all containers on the node are healthy. It takes care of starting, stopping, and maintaining application containers organized into pods as directed by the control plane.
    - Kubelet monitors the state of a pod, and if not in the desired state, the pod re-deploys to the same node. Node status is relayed every few seconds via heartbeat messages to the master. Once the master detects a node failure, the Replication Controller observes this state change and launches pods on other healthy nodes.
  - Kube-proxy: The Kube-proxy is an implementation of a network proxy and a load balancer, and it supports the service abstraction along with other networking operation. It is responsible for routing traffic to the appropriate container based on IP and port number of the incoming request.
  - Container runtime: A container resides inside a pod. The container is the lowest level of a micro-service that holds the running application, libraries, and their dependencies. Containers can be exposed to the world through an external IP address. Kubernetes supports Docker containers since its first version, and in July 2016 rkt container engine was added.

Kubernetes architecture and components

- Add-ons
  - Add-ons operate just like any other application running within the cluster: they are implemented via pods and services, and are only different in that they implement features of the Kubernetes cluster.
  - The pods may be managed by Deployments, ReplicationControllers, and so on.
  - There are many add-ons, and the list is growing. Some of the more important are:
    - DNS: All Kubernetes clusters should have cluster DNS; it is a mandatory feature. Cluster DNS is a DNS server, in addition to the other DNS server(s) in your environment, which serves DNS records for Kubernetes services. Containers started by Kubernetes automatically include this DNS server in their DNS searches.
    - Web UI: This is a general purpose, web-based UI for Kubernetes clusters. It allows users to manage and troubleshoot applications running in the cluster, as well as the cluster itself.
  - Container Resource Monitoring:
    - Container Resource Monitoring provides this capability by recording metrics about containers in a central database, and provides a UI for browsing that data. The cAdvisor is a component on a slave node that provides a limited metric monitoring capability.
  - Cluster-level logging:
    - Logs should have a separate storage and lifecycle independent of nodes, pods, or containers. Otherwise, node or pod failures can cause loss of event data. The ability to do this is called cluster-level logging, and such mechanisms are responsible for saving container logs to a central log store with search/browsing interface.
    - Kubernetes provides no native storage solution for log data, but one can integrate many existing logging solutions into the Kubernetes cluster.

Kubernetes Installation and Configuration

- Deciding where to run Kubernetes depends on what resources you have available and how much flexibility you need.
- You can run Kubernetes almost anywhere, from your laptop to VMs on a cloud provider to a rack of bare metal servers.
- You can also set up a fully-managed cluster by running a single command or craft your own customized cluster on your bare metal servers.
  - Local-machine Solutions
    - A local-machine solution is an easy way to get started with Kubernetes. You can create and test Kubernetes clusters without worrying about consuming cloud resources and quotas.
    - You should pick a local solution if you want to:
      - Try or start learning about Kubernetes
      - Develop and test clusters locally
    - Minikube is a method for creating a local, single-node Kubernetes cluster for development and testing. Setup is completely automated and doesn't require a cloud provider account.
    - Kubeadm-dind is a multi-node (while minikube is single-node) Kubernetes cluster which only requires a docker daemon. It uses docker-in-docker technique to spawn the Kubernetes cluster.

Kubernetes Installation and Configuration

- Hosted Solutions
  - Hosted solutions are a convenient way to create and maintain Kubernetes clusters. They manage and operate your clusters so you don't have to.
    - You should pick a hosted solution if you:
    - Want a fully-managed solution
    - Want to focus on developing your apps or services
    - Don't have dedicated site reliability engineering (SRE) team but want high availability
    - Don't have resources to host and monitor your clusters
  - Refer url - https://kubernetes.io/docs/setup/pick-right-solution/#hosted-solutions
- Turnkey – Cloud Solutions
  - These solutions allow you to create Kubernetes clusters with only a few commands and are actively developed and have active community support.
  - They can also be hosted on a range of Cloud IaaS providers, but they offer more freedom and flexibility in exchange for effort.
  - You should pick a turnkey cloud solution if you:
    - Want more control over your clusters than the hosted solutions allow
    - Want to take on more operations ownership
  - Refer url -https://kubernetes.io/docs/setup/pick-right-solution/#turnkey-cloud-solutions

Kubernetes Installation and Configuration

- Turnkey – On-Premises Solutions
  - These solutions allow you to create Kubernetes clusters on your internal, secure, cloud network with only a few commands.
  - You should pick a on-prem turnkey cloud solution if you:
    - Want to deploy clusters on your private cloud network
    - Have a dedicated SRE team
    - Have the resources to host and monitor your clusters
  - Refer url -https://kubernetes.io/docs/setup/pick-right-solution/#on-premises-turnkey-cloud-solutions

- Custom Solutions
  - Custom solutions give you the most freedom over your clusters but require the most expertise.
  - These solutions range from bare-metal to cloud providers on different operating systems.
  - Refer url - https://kubernetes.io/docs/setup/pick-right-solution/#custom-solutions

Create a Kubernetes simple environment with Minikube

- Minikube is a tool that makes it easy to run Kubernetes locally.
- Minikube runs a single-node Kubernetes cluster inside a VM on your laptop for users looking to try out Kubernetes or develop with it day-to-day.
- Minikube supports Kubernetes features such as:
  - DNS
  - NodePorts
  - ConfigMaps and Secrets
  - Dashboards
  - Container Runtime: Docker, rkt, CRI-O and containerd
  - Enabling CNI (Container Network Interface)
  - Ingress

Create a Kubernetes simple environment with Minikube

- Installation
  - Windows manual installation
    - To install Minikube manually on Windows, download minikube-windows-amd64, rename it to minikube.exe, and add it to your path.
    - Refer url - https://github.com/kubernetes/minikube/releases/latest
  - Windows Installer
    - To install Minikube manually on windows using Windows Installer, download minikube-installer.exe and execute the installer.
    - Refer url - https://docs.microsoft.com/en-us/windows/desktop/msi/windows-installer-portal
  - Install a Hypervisor
    - macOS VirtualBox, VMware Fusion, HyperKit
    - Linux VirtualBox, KVM
    - Windows VirtualBox, Hyper-V
  - Install kubectl
    - Install kubectl according to the instructions in Install and Set Up kubectl (https://kubernetes.io/docs/tasks/tools/install-kubectl/).

- Refer –for minikube on linux - https://kubernetes.io/docs/tasks/tools/install-minikube/, https://linuxhint.com/install-minikube-ubuntu/ , https://www.radishlogic.com/kubernetes/running-minikube-in-aws-ec2-ubuntu/

Create a Kubernetes simple environment with Minikube

- Working with Minikube
  - minikube start - This will start minikube
  - minikube dashboard - This will open dashboard in the browser
  - Use the kubectl create command to create a Deployment that manages a Pod. The Pod runs a Container based on the provided Docker image.
    - kubectl create deployment hello-node --image=gcr.io/hello-minikube-zero-install/hello-node
  - View the Deployment:
    - kubectl get deployments
  - View the Pod:
    - kubectl get pods
  - View cluster events:
    - kubectl get events
  - View the kubectl configuration
    - kubectl config view

Create a Kubernetes simple environment with Minikube

Create Service

- By default, the Pod is only accessible by its internal IP address within the Kubernetes cluster. To make the hello-node Container accessible from outside the Kubernetes virtual network, you have to expose the Pod as a Kubernetes Service.
- Expose the Pod to the public internet using the kubectl expose command:
  - kubectl expose deployment hello-node --type=LoadBalancer --port=8080
  - The --type=LoadBalancer flag indicates that you want to expose your Service outside of the cluster.
- View the Service you just created:
  - kubectl get services
- On Minikube, the LoadBalancer type makes the Service accessible through the minikube service command.
- Run the following command:
  - minikube service hello-node
  - This opens up a browser window that serves your app and shows the “Hello World” message.

Cluster management- the Kubectl cli and API structure

- Kubectl is a command line interface for running commands against Kubernetes clusters.
- Refer - https://kubernetes.io/docs/reference/kubectl/overview/

- Kubernetes API Overview
  - The REST API is the fundamental fabric of Kubernetes. All operations and communications between components, and external user commands are REST API calls that the API Server handles.

- Client Libraries
  - To write applications using the Kubernetes REST API, you can use a client library for the programming language you are using.
    - Go github.com/kubernetes/client-go/
    - Python github.com/kubernetes-client/python/
    - Java github.com/kubernetes-client/java
    - dotnet github.com/kubernetes-client/csharp
    - JavaScript github.com/kubernetes-client/javascript

Kubernetes Web UI – Kubernetes dashboard

- Dashboard is a web-based Kubernetes user interface. You can use Dashboard to deploy containerized applications to a Kubernetes cluster, troubleshoot your containerized application, and manage the cluster resources.
- You can use Dashboard to get an overview of applications running on your cluster, as well as for creating or modifying individual Kubernetes resources (such as Deployments, Jobs, DaemonSets, etc).
- For example, you can scale a Deployment, initiate a rolling update, restart a pod or deploy new applications using a deploy wizard.
- Dashboard also provides information on the state of Kubernetes resources in your cluster and on any errors that may have occurred.

Kubernetes Web UI – Kubernetes dashboard

- Deploying the Dashboard UI
  - The Dashboard UI is not deployed by default. To deploy it, run the following command:
  - kubectl create -f https://raw.githubusercontent.com/kubernetes/dashboard/master/aio/deploy/recommended/kubernetes-dashboard.yaml
- Accessing the Dashboard UI
  - You can access Dashboard using the kubectl command-line tool by running the following command:

    kubectl proxy
  - Kubectl will make Dashboard available at
  - http://localhost:8001/api/v1/namespaces/kube-system/services/https:kubernetes-dashboard:/proxy/.
  - The UI can only be accessed from the machine where the command is executed.
  - See kubectl proxy --help for more options.

Working with Kubernetes Objects

- Kubernetes Objects are persistent entities in the Kubernetes system.Kubernetes uses these entities to represent the state of your cluster.
- A Kubernetes object is a “record of intent”–once you create the object, the Kubernetes system will constantly work to ensure that object exists. By creating an object, you’re effectively telling the Kubernetes system what you want your cluster’s workload to look like; this is your cluster’s desired state.
- To work with Kubernetes objects–whether to create, modify, or delete them–you’ll need to use the Kubernetes API.
- When you use the kubectl command-line interface, for example, the CLI makes the necessary Kubernetes API calls for you.
- You can also use the Kubernetes API directly in your own programs using one of the Client Libraries.

Working with Kubernetes Objects

- Describing a Kubernetes Object
  - When you create an object in Kubernetes, you must provide the object spec that describes its desired state, as well as some basic information about the object (such as a name).
  - When you use the Kubernetes API to create the object (either directly or via kubectl), that API request must include that information as JSON in the request body.
  - Most often, you provide the information to kubectl in a .yaml file. kubectl converts the information to JSON when making the API request.
- To create deployment object using a .yaml
  - kubectl create -f https://k8s.io/examples/application/deployment.yaml --record

```
apiVersion: apps/v1 # for versions before 1.9.0 use apps/v1beta2
kind: Deployment
metadata:
  name: nginx-deployment
spec:
  selector:
    matchLabels:
      app: nginx
  replicas: 2 # tells deployment to run 2 pods matching the template
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx:1.7.9
        ports:
        - containerPort: 80
```

Working with Kubernetes Objects

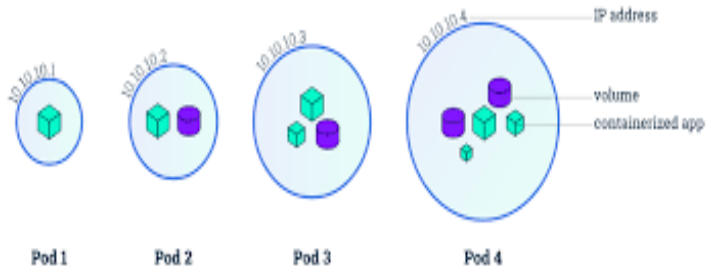
- The output is similar to this:
  - deployment.apps/nginx-deployment created
- Required Fields
  - In the .yaml file for the Kubernetes object you want to create, you'll need to set values for the following fields:
    - apiVersion - Which version of the Kubernetes API you're using to create this object
    - kind - What kind of object you want to create
    - metadata - Data that helps uniquely identify the object, including a name string, UID, and optional namespace
    - You'll also need to provide the object spec field.
    - The precise format of the object spec is different for every Kubernetes object, and contains nested fields specific to that object.
    - The Kubernetes API Reference can help you find the spec format for all of the objects you can create using Kubernetes.
    - For example, the spec format for a Pod object can be found here, and the spec format

Pods

- A pod is a group of one or more containers, with shared storage/network, and a specification for how to run the containers.
- A pod models an application-specific “logical host” - it contains one or more application containers which are relatively tightly coupled — in a pre-container world, being executed on the same physical or virtual machine would mean being executed on the same logical host.
- Containers within a pod share an IP address and port space, and can find each other via localhost. They can also communicate with each other using standard inter-process communications commands.
- Containers in different pods have distinct IP addresses and can not communicate by IPC without special configuration. These containers usually communicate with each other via Pod IP addresses.

Pods

- Like individual application containers, pods are considered to be relatively ephemeral (rather than durable) entities.
- If a node dies, the pods scheduled to that node are scheduled for deletion, after a timeout period.
- When something is said to have the same lifetime as a pod, such as a volume, that means that it exists as long as that pod (with that UID) exists.
- If that pod is deleted for any reason, even if an identical replacement is created, the related thing (e.g. volume) is also destroyed and created anew.



Pods

- Life cycle
  - Here are the possible values for phase:

Value	Description
`Pending`	The Pod has been accepted by the Kubernetes system, but one or more of the Container images has not been created. This includes time before being scheduled as well as time spent downloading images over the network, which could take a while.
`Running`	The Pod has been bound to a node, and all of the Containers have been created. At least one Container is still running, or is in the process of starting or restarting.
`Succeeded`	All Containers in the Pod have terminated in success, and will not be restarted.
`Failed`	All Containers in the Pod have terminated, and at least one Container has terminated in failure. That is, the Container either exited with non-zero status or was terminated by the system.
`Unknown`	For some reason the state of the Pod could not be obtained, typically due to an error in communicating with the host of the Pod.
`Completed`	The pod has run to completion as there's nothing to keep it running eg. Completed Jobs.
`CrashLoopBackOff`	This means that one of the containers in the pod has exited unexpectedly, and perhaps with a non-zero error code even after restarting due to restart policy.

Pods

- Pods configuration
  - Kubectl create –f pod-nginx.yaml => this will create the Pod
  - Pod-nginx.yaml

```
apiVersion: v1
kind: Pod
metadata:
  name: nginx
  labels:
    name: nginx
spec:
  containers:
  - name: nginx
    image: nginx
    ports:
    - containerPort: 80
```

Refer - https://kubernetes.io/docs/tasks/configure-pod-container/configure-pod-initialization/

Deployments

- A Deployment controller provides declarative updates for Pods and ReplicaSets.
- You describe a desired state in a Deployment object, and the Deployment controller changes the actual state to the desired state at a controlled rate.
- You can define Deployments to create new ReplicaSets, or to remove existing Deployments and adopt all their resources with new Deployments.

Deployments

- The following are typical use cases for Deployments:

  - Create a Deployment to rollout a ReplicaSet. The ReplicaSet creates Pods in the background. Check the status of the rollout to see if it succeeds or not.
  - Declare the new state of the Pods by updating the PodTemplateSpec of the Deployment. A new ReplicaSet is created and the Deployment manages moving the Pods from the old ReplicaSet to the new one at a controlled rate. Each new ReplicaSet updates the revision of the Deployment.
  - Rollback to an earlier Deployment revision if the current state of the Deployment is not stable. Each rollback updates the revision of the Deployment.
  - Scale up the Deployment to facilitate more load.
  - Pause the Deployment to apply multiple fixes to its PodTemplateSpec and then resume it to start a new rollout.
  - Use the status of the Deployment as an indicator that a rollout has stuck.
  - Clean up older ReplicaSets that you don’t need anymore.

Deployments

- The Deployment creates three replicated Pods, indicated by the replicas field.
- The selector field defines how the Deployment finds which Pods to manage.
  - In this case, you simply select a label that is defined in the Pod template (app: nginx) (more sophisticated selection rules are possible, as long as the Pod template itself satisfies the rule)
  - Note: matchLabels is a map of {key,value} pairs. A single {key,value} in the matchLabels map is equivalent to an element of matchExpressions, whose key field is "key", the operator is "In", and the values array contains only "value". The requirements are ANDed.

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
  labels:
    app: nginx
spec:
  replicas: 3
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx:1.7.9
        ports:
        - containerPort: 80
```

- Refer- https://kubernetes.io/docs/concepts/workloads/controllers/deployment/

Deployments

- The template field contains the following sub-fields:
  - The Pods are labeled app: nginxusing the labels field.
  - The Pod template's specification, or .template.spec field, indicates that the Pods run one container, nginx, which runs the nginx Docker Hub image at version 1.7.9.
  - Create one container and name it nginx using the name field.
  - Run the nginx image at version 1.7.9.
  - Open port 80 so that the container can send and accept traffic.
- To create this Deployment, run the following command:
  - kubectl create -f ./nginx-deployment.yaml

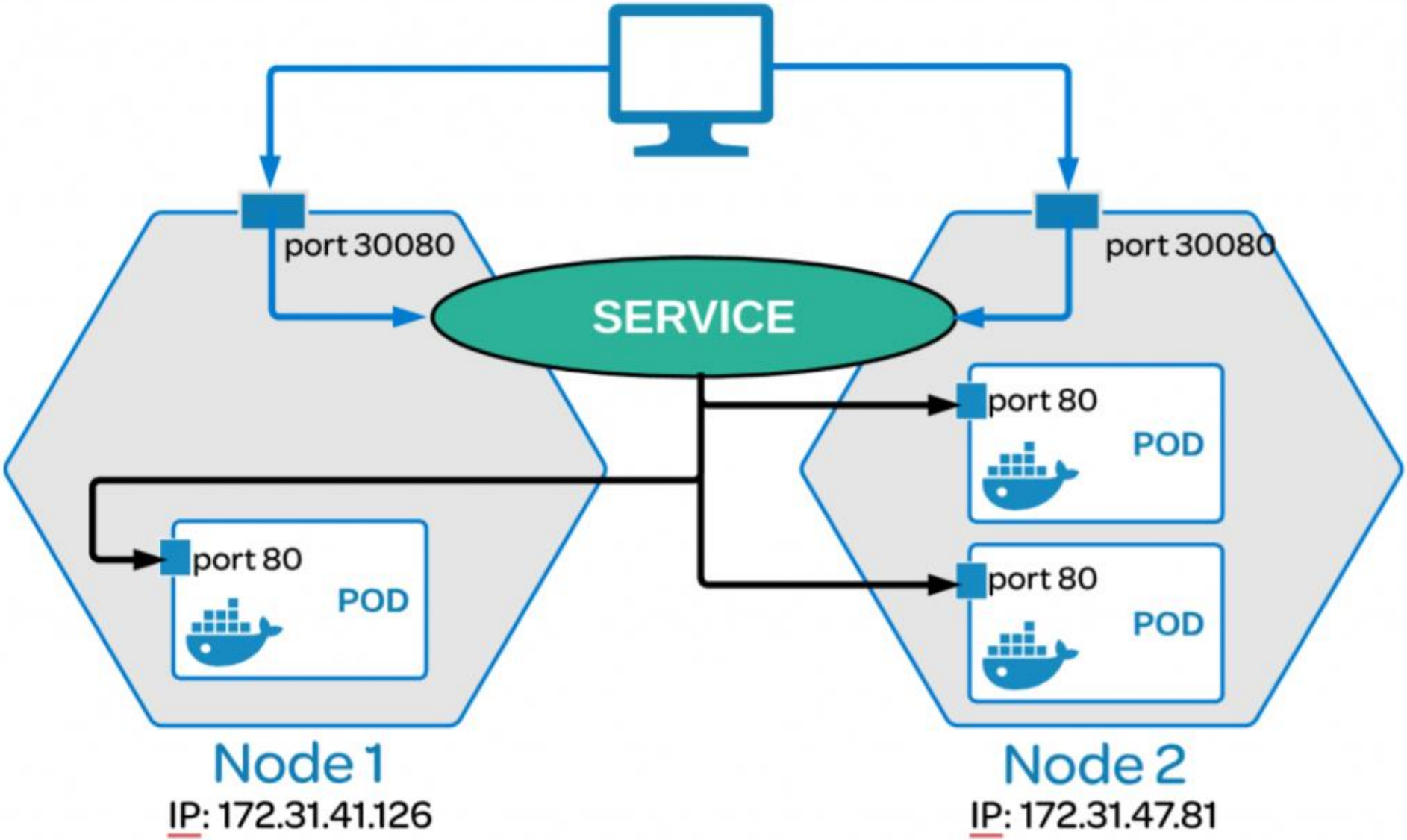
```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
  labels:
    app: nginx
spec:
  replicas: 3
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx:1.7.9
        ports:
        - containerPort: 80
```

Services & Service types

- A Kubernetes Service is an abstraction which defines a logical set of Pods and a policy by which to access them - sometimes called a micro-service.
- The set of Pods targeted by a Service is (usually) determined by a Label Selector.
- A Service in Kubernetes is a REST object, similar to a Pod.
- Like all of the REST objects, a Service definition can be POSTed to the apiserver to create a new instance.
- For example, suppose you have a set of Pods that each expose port 9376 and carry a label "app=MyApp".

Kubernetes Service

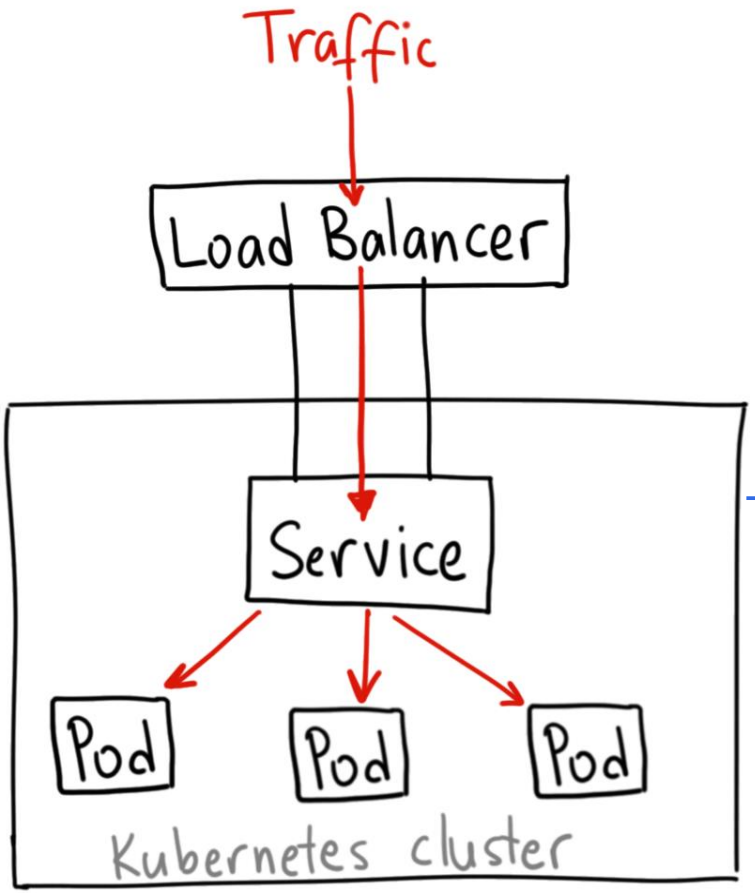
A service allows you to dynamically access a group of replica pods.

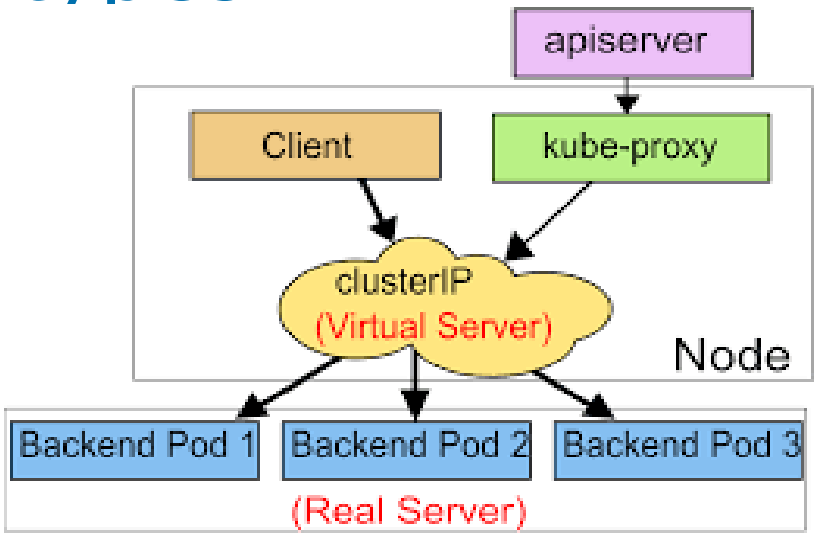


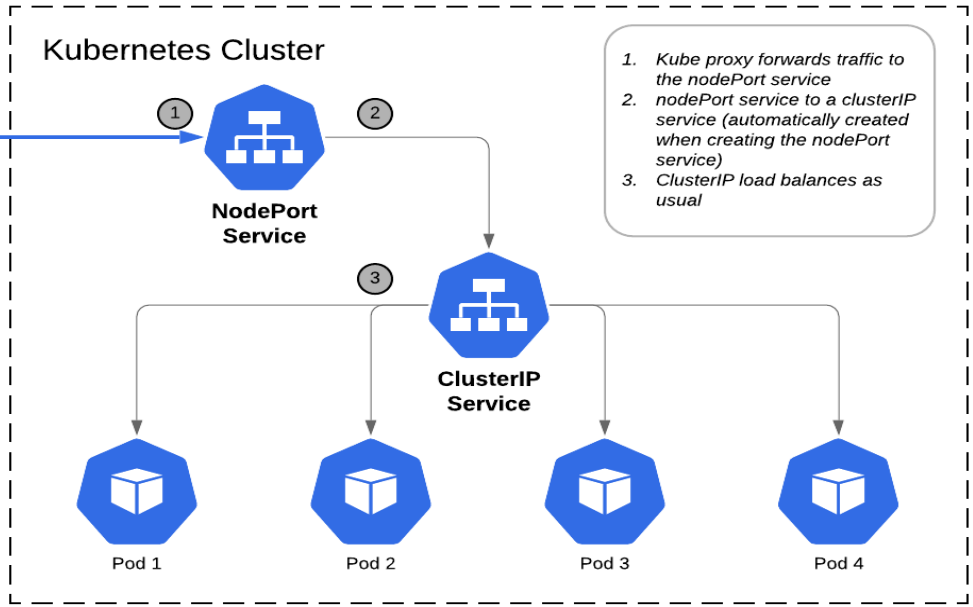
Services & Service types

- Publishing services - service types
  - 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.
  - NodePort: Exposes the service on each Node's IP at a static port (the NodePort). A ClusterIP service, to which the NodePort service will route, is automatically created. You'll be able to contact the NodePort service, from outside the cluster, by requesting <NodeIP>:<NodePort>.
  - LoadBalancer: Exposes the service externally using a cloud provider's load balancer. NodePort and ClusterIP services, to which the external load balancer will route, 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. This requires version 1.7 or higher of kube-dns.

Services & Service types







Services & Service types

- Example

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

```
---
apiVersion: v1
kind: Service
metadata:
  name: nginx
  namespace: default
  labels:
    app: nginx
spec:
ports:
  - name: http
    port: 80
    protocol: TCP
    targetPort: 80
  selector:
    app: nginx
  type: NodePort
```

Refer - https://kubernetes.io/docs/concepts/services-networking/connect-applications-service/

Replica Sets

- A ReplicaSet’s purpose is to maintain a stable set of replica Pods running at any given time. As such, it is often used to guarantee the availability of a specified number of identical Pods.
- A ReplicaSet is defined with fields, including a selector that specifies how to identify Pods it can acquire, a number of replicas indicating how many Pods it should be maintaining, and a pod template specifying the data of new Pods it should create to meet the number of replicas criteria.
- A ReplicaSet then fulfills its purpose by creating and deleting Pods as needed to reach the desired number. When a ReplicaSet needs to create new Pods, it uses its Pod template.
- Deployment is a higher-level concept that manages ReplicaSets and provides declarative updates to Pods along with a lot of other useful features. Therefore, it is recommend to use Deployments instead of directly using ReplicaSets, unless you require custom update orchestration.

Replication Controllers and Deployment

- A Deployment that configures a ReplicaSet is now the recommended way to set up replication.
- A ReplicationController ensures that a specified number of pod replicas are running at any one time.
- In other words, a ReplicationController makes sure that a pod or a homogeneous set of pods is always up and available.

- Refer-https://kubernetes.io/docs/concepts/workloads/controllers/replicationcontroller/

Accessing shell of running container

- Use kubectl exec to get a shell to a running Container
- Create the Pod:
  - kubectl create -f ./shell-demo.yaml
- Verify that the Container is running:
  - kubectl get pod shell-demo
- Get a shell to the running Container:
  - kubectl exec -it shell-demo -- /bin/bash
- Running individual commands in a Container
  - In an ordinary command window, not your shell, list the environment variables in the running Container:
  - kubectl exec shell-demo env

```
apiVersion: v1
kind: Pod
metadata:
  name: shell-demo
spec:
  volumes:
  - name: shared-data
    emptyDir: {}
  containers:
  - name: nginx
    image: nginx
    volumeMounts:
    - name: shared-data
      mountPath: /usr/share/nginx/html
```

Accessing shell of running container

- Use kubectl exec to get a shell to a running Container
- Opening a shell when a Pod has more than one Container
  - If a Pod has more than one Container, use --container or -c to specify a Container in the kubectl exec command.
  - For example, suppose you have a Pod named my-pod, and the Pod has two containers named main-app and helper-app.
  - The following command would open a shell to the main-app Container.

```
kubectl exec -it my-pod --container main-app -- /bin/bash
```

Kubernetes networking

- Every Pod gets its own IP address. This creates a clean, backwards-compatible model where Pods can be treated much like VMs or physical hosts from the perspectives of port allocation, naming, service discovery, load balancing, application configuration, and migration.
- Pods on a node can communicate with all pods on all nodes without NAT
- Agents on a node (e.g. system daemons, kubelet) can communicate with all pods on that node
- Kubernetes IP addresses exist at the Pod scope - containers within a Pod share their network namespaces - including their IP address.
- There are a number of ways that this network model can be implemented(Refer -https://kubernetes.io/docs/concepts/cluster-administration/networking/)

Kubernetes networking

- The Kubernetes network model
- Every Pod gets its own IP address. This creates a clean, backwards-compatible model where Pods can be treated much like VMs or physical hosts from the perspectives of port allocation, naming, service discovery, load balancing, application configuration, and migration.
- Kubernetes imposes the following fundamental requirements on any networking implementation (barring any intentional network segmentation policies):
- pods on a node can communicate with all pods on all nodes without NAT
- agents on a node (e.g. system daemons, kubelet) can communicate with all pods on that node
- Kubernetes IP addresses exist at the Pod scope - containers within a Pod share their network namespaces - including their IP address.
- There are a number of ways that this network model can be implemented(Refer -https://kubernetes.io/docs/concepts/cluster-administration/networking/)

Service discovery

- Service discovery is the process of figuring out how to connect to a service.
- Kubernetes service discovery find services through two approaches:
- Using the environment variables that use the same conventions as those created by Docker links.
  - The easiest way to find out what environment variables are exposed are to exec the env command within a pod.
- Using DNS to resolve the service names to the service's IP address.
  - Kubernetes has a kube-dns addon that exposes the service's name as a DNS entry. As a result, you can tell your application to connect to a host name.
  - The advantage of this approach is that you do not need to do anything different than you would otherwise.
  - (Refer- https://kubernetes.io/docs/concepts/services-networking/dns-pod-service/)

Volumes and Application Data

- On-disk files in a Container are ephemeral, which presents some problems for non-trivial applications when running in Containers.
- First, when a Container crashes, kubelet will restart it, but the files will be lost - the Container starts with a clean state.
- Second, when running Containers together in a Pod it is often necessary to share files between those Containers.
- The Kubernetes Volume abstraction solves both of these problems.
- Volume outlives any Containers that run within the Pod, and data is preserved across Container restarts.
- Type of volumes
  - emptyDir - An emptyDir volume is first created when a Pod is assigned to a Node, and exists as long as that Pod is running on that node.
    - As the name says, it is initially empty.
    - Containers in the Pod can all read and write the same files in the emptyDir volume, though that volume can be mounted at the same or different paths in each Container.
    - When a Pod is removed from a node for any reason, the data in the emptyDir is deleted forever.

Volumes and Application Data

- Type of volumes
  - emptyDir
    - Some uses for an emptyDir are:
      - scratch space, such as for a disk-based merge sort
      - checkpointing a long computation for recovery from crashes
      - holding files that a content-manager Container fetches while a webserver Container serves the data

```
apiVersion: v1
kind: Pod
metadata:
  name: test-pd
spec:
  containers:
  - image: k8s.gcr.io/test-webserver
    name: test-container
    volumeMounts:
    - mountPath: /cache
      name: cache-volume
  volumes:
  - name: cache-volume
    emptyDir: {}
```

- Refer - https://kubernetes.io/docs/tasks/configure-pod-container/configure-volume-storage/

Volumes and Application Data

- Type of volumes
  - HostPath
    - A hostPath volume mounts a file or directory from the host node's filesystem into your Pod.
    - For example, some uses for a hostPath are:
      - running a Container that needs access to Docker internals; use a hostPath of /var/lib/docker
      - running cAdvisor in a Container; use a hostPath of /sys
      - allowing a Pod to specify whether a given hostPath should exist prior to the Pod running, whether it should be created, and what it should exist as

```
apiVersion: v1
kind: Pod
metadata:
  name: test-pd
spec:
  containers:
  - image: k8s.gcr.io/test-webserver
    name: test-container
    volumeMounts:
    - mountPath: /test-pd
      name: test-volume
  volumes:
  - name: test-volume
    hostPath:
      # directory location on host
      path: /data
      # this field is optional
      type: Directory
```

https://kubernetes.io/docs/tasks/configure-pod-container/configure-persistent-volume-storage/

Logging

- Application and systems logs can help you understand what is happening inside your cluster. The logs are particularly useful for debugging problems and monitoring cluster activity.
- Most modern applications have some kind of logging mechanism; as such, most container engines are likewise designed to support some kind of logging.
- The easiest and most embraced logging method for containerized applications is to write to the standard output and standard error streams.

Logging

- To run this pod, use the following command:
- kubectl create -f https://k8s.io/examples/debug/counter-pod.yaml
- To fetch the logs, use the kubectl logs command, as follows:

kubectl logs counter

```
apiVersion: v1
kind: Pod
metadata:
  name: counter
spec:
  containers:
  - name: count
    image: busybox
    args: [/bin/sh, -c,
            'i=0; while true; do echo "$i:
$(date)"; i=$((i+1)); sleep 1; done']
```

Health Checks

- The kubelet uses liveness probes to know when to restart a Container.
- For example, liveness probes could catch a deadlock, where an application is running, but unable to make progress.
- Restarting a Container in such a state can help to make the application more available despite bugs.
- The kubelet uses readiness probes to know when a Container is ready to start accepting traffic.
- A Pod is considered ready when all of its Containers are ready. One use of this signal is to control which Pods are used as backends for Services.
- When a Pod is not ready, it is removed from Service load balancers.

Refer- https://kubernetes.io/docs/tasks/configure-pod-container/configure-liveness-readiness-probes/

Rolling Updates

- Users expect applications to be available all the time and developers are expected to deploy new versions of them several times a day.
- In Kubernetes this is done with rolling updates. Rolling updates allow Deployments' update to take place with zero downtime by incrementally updating Pods instances with new ones.
- The new Pods will be scheduled on Nodes with available resources.
- A rolling update applies changes to the configuration of pods being managed by a replication controller. The changes can be passed as a new replication controller configuration file; or, if only updating the image, a new container image can be specified directly.
- A rolling update works by:
  - Creating a new replication controller with the updated configuration.
  - Increasing/decreasing the replica count on the new and old controllers until the correct number of replicas is reached.
  - Deleting the original replication controller.
- Rolling updates are initiated with - kubectl rolling-update command
- Refer-https://kubernetes.io/docs/tasks/run-application/rolling-update-replication-controller/

Kubernetes Cheat Sheet

What is Kubernetes?

Kubernetes is a platform for managing containerized workloads. Kubernetes orchestrates computing, networking and storage to provide a seamless portability across infrastructure providers.

Viewing Resource Information

Nodes

```
$ kubectl get no
$ kubectl get no -o wide
$ kubectl describe no
$ kubectl get no -o yaml
$ kubectl get node --selector=[label_name]
$ kubectl get nodes -o jsonpath='{.items[*].status.addresses[?(@.type=="ExternalIP")].address}'
$ kubectl top node [node_name]
```

Pods

```
$ kubectl get po
$ kubectl get po -o wide
$ kubectl describe po
$ kubectl get po --show-labels
$ kubectl get po -l app=nginx
$ kubectl get po -o yaml
$ kubectl get pod [pod_name] -o yaml --export
$ kubectl get pod [pod_name] -o yaml --export > nameoffile.yaml
$ kubectl get pods --field-selector status.phase=Running
```

Namespaces

```
$ kubectl get ns
$ kubectl get ns -o yaml
$ kubectl describe ns
```

Deployments

```
$ kubectl get deploy
$ kubectl describe deploy
$ kubectl get deploy -o wide
$ kubectl get deploy -o yaml
```

Services

```
$ kubectl get svc
$ kubectl describe svc
$ kubectl get svc -o wide
$ kubectl get svc -o yaml
$ kubectl get svc --show-labels
```

DaemonSets

```
$ kubectl get ds
$ kubectl get ds --all-namespaces
$ kubectl describe ds [daemonset_name] -n [namespace_name]
$ kubectl get ds [ds_name] -n [ns_name] -o yaml
```

Events

```
$ kubectl get events
$ kubectl get events -n kube-system
$ kubectl get events -w
```

Logs

```
$ kubectl logs [pod_name]
$ kubectl logs --since=1h [pod_name]
$ kubectl logs --tail=20 [pod_name]
$ kubectl logs -f -c [container_name] [pod_name]
$ kubectl logs [pod_name] > pod.log
```

Service Accounts

```
$ kubectl get sa
$ kubectl get sa -o yaml
$ kubectl get serviceaccounts default -o yaml > ./sa.yaml
$ kubectl replace serviceaccount default -f ./sa.yaml
```

ReplicaSets

```
$ kubectl get rs
$ kubectl describe rs
$ kubectl get rs -o wide
$ kubectl get rs -o yaml
```

Roles

```
$ kubectl get roles --all-namespaces
$ kubectl get roles --all-namespaces -o yaml
```

Secrets

```
$ kubectl get secrets
$ kubectl get secrets --all-namespaces
$ kubectl get secrets -o yaml
```

ConfigMaps

```
$ kubectl get cm
$ kubectl get cm --all-namespaces
$ kubectl get cm --all-namespaces -o yaml
```

Ingress

```
$ kubectl get ing
$ kubectl get ing --all-namespaces
```

PersistentVolume

```
$ kubectl get pv
$ kubectl describe pv
```

PersistentVolumeClaim

```
$ kubectl get pvc
$ kubectl describe pvc
```

Summary

- You learned Kubernetes basics and how to use it for automating deployment, scaling, and management of containerized applications.
- Now you can use it in your project for automating deployments and management of containerized applications.

People matter, results count.

About Capgemini

A global leader in consulting, technology services and digital transformation, Capgemini is at the forefront of innovation to address the entire breadth of clients' opportunities in the evolving world of cloud, digital and platforms. Building on its strong 50-year heritage and deep industry-specific expertise, Capgemini enables organizations to realize their business ambitions through an array of services from strategy to operations. Capgemini is driven by the conviction that the business value of technology comes from and through people. It is a multicultural company of 200,000 team members in over 40 countries. The Group reported 2017 global revenues of EUR 12.8 billion.

Learn more about us at

www.capgemini.com