# Kubernetes Essentials

About CloudThat

* CloudThat is the first company in India to Cloud Training & Consulting services for mid-market & enterprise clients around the world. With expertise in major Cloud platforms including Microsoft Azure, Amazon Web Services (AWS) and Google Cloud Platform (GCP). CloudThat is uniquely positioned to be the single technology source for organizations looking to utilize the flexibility and power Cloud Computing provides.
* CloudThat is focused on quickly empowering IT professionals and organizations with leveraging Cloud, Big Data & IoT. Founded by Bhavesh Goswami, an ex-Microsoft and ex-Amazonian who was part of the Microsoft and AWS product development teams.
* Till date we have trained more than 200,000 IT professionals and conducted corporate training for some of the fortune 500 companies which include Accenture, Infosys, Fidelity, HCL, Intuit, GE, TCS, HP, SAP, Oracle, Western Union, Philips, Flipkart, L&T and Samsung, just to name a few.
* We have presence in Bengaluru, USA & UK, but offer on-site and pre-scheduled public batches in different IT centric cities of India and Overseas.
* CloudThat is a Microsoft Gold Partner, Advanced AWS Consulting partner, Google Consulting Partner, Red Hat Certified Training Partner, MongoDB Ready Partner, and part of Pearson Testing Network.
* Our current course offerings are on Azure, Dynamics 365, Microsoft Security Suite, AI & Machine Learning, Cloud Security, Analytics, Red Hat, IoT, DevOps, Chef, Docker, Ansible, Kubernetes, Blockchain, Big Data, etc. We are constantly adding more courses and more consulting offerings.

About the Author

**Bhavesh Goswami,**

**The Co-Founder & CEO of CloudThat,**

CloudThat is a leading expert in Cloud Computing space with over a decade of experience. He was in the initial development team of Amazon Simple Storage Service (S3) at Amazon Web Services (AWS) in Seattle. He honed his Cloud Computing skills at Amazon, where he helped ship the first version of S3 in 2006. Later he moved to Microsoft after over three years at Amazon to take up the challenge to help manage Cosmos, the Cloud storage, and Big Data computational engine that power all Microsoft’s Online Services, including Bing.

In 2010, after living in the USA for ten years, he came to India in search of a challenge. He realized that the rapidly changing technology landscape and the busy schedule of IT professionals were not conducive to acquire new skills. Thus, he started CloudThat, a company focused on quickly empowering professionals & corporates on Cloud & Big Data through training & consulting services. He has personally trained over 1000 people on various Cloud technologies like AWS, Microsoft Azure, Google App Engine and more since early 2012.

Bhavesh has spoken at various Cloud and Big Data conferences and events like ‘7th Cloud Computing & Big Data’ and has been the Keynote Speaker at ‘International Conference on Computer Communication and Informatics’. He has authored numerous research papers and patents in various fields.

Bhavesh earned his MS in Computer Science from the University of South Florida, where he was awarded Scholarships for Teaching Assistantship and Research Assistantship in 2003.

**Haris AK**

**Cloud Solutions Architect - DevOps, CloudThat**

Haris works as Cloud Solutions Architect in CloudThat technologies, being passionate about ever evolving technology he works on Docker, Kubernetes, and DevOps Technologies.

Haris architects’ solutions on Cloud as well on-Premises using wide array of platforms/technologies like AWS, Azure and VMWare. Having core training and consulting experience, he is involved in delivering Git, Jenkins, Ansible, Docker, Kubernetes training to corporates like Infosys, Wipro, TCS, Accenture, NetApp, EY, Optym, Valtech etc.

Copyright

**Copyright @2012-21, CloudThat. All Rights Reserved.**

The contents of this course, its modules, and its related materials, including handouts to participants, are Copyright @2012-21, CloudThat. If you are using this curriculum, you are agreeing to comply with and be bound by the following terms and conditions of use, which together with our privacy policy govern ‘CloudThat’ relationship with the participant in relation to this curriculum.

No part of this curriculum may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, electronic, mechanical, recording or otherwise, without the prior written permission of CloudThat. Please note that the content in this curriculum is protected under copyright law.

The content of this curriculum cannot use in part or whole for conducting any form of training without written consent of CloudThat. The content of this guide is provided for informational use only and is subject to change without notice and should not be rendered as a commitment by CloudThat. CloudThat assumes no responsibility or liability for any errors or inaccuracies that may appear in the informational content contained in this curriculum.

Please remember this curriculum contains material, which is owned by or licensed to us, any existing artwork, or images that you may want to include in your project may be protected under copyright law. The unauthorized incorporation of such material into your new work could be a violation of the rights of the copyright owner. Please be sure to obtain any permission required from the copyright owner.

Any references to company names in sample files are for demonstration purposes only and are not intended to refer to any actual organization. Please remember that use of any information of this curriculum is entirely at the personal expense of the participant for which CloudThat will not be liable. The participant is responsible for ensuring that any products, services, or information available through this curriculum meet their specific requirements.

All trademarks reproduced in this curriculum, which is not the property of, or licensed to the operator, are acknowledged in the curriculum. From time to time, this curriculum may also include links to other websites. These links are provided for the participant’s convenience to provide further information. They do not signify that CloudThat endorses the website(s). CloudThat has no responsibility for the content of the linked website(s).

Unauthorized use of this curriculum will give a right to CloudThat to a claim for damages and/or be a criminal offense.

# Chapter One: Fundamentals of Kubernetes

**Topics covered in this unit:**

* Introduction to Kubernetes
* Kubernetes Components
* Why Containers

## Introduction to Kubernetes

Kubernetes is an open-source platform used to automate container operations. It is used to automate deployment, scaling and other container management tasks across the varied infrastructure. It is an orchestration tool used with container clusters. These clusters can span hosts across public, private or hybrid clouds.

It was initially designed and developed by Google and was called “k8s”. Google has since been contributing massively to Linux container technology.

It is extremely portable, and it can be run on the bare-metal environment, cloud platforms or a combination of the two.

**Need for Kubernetes**

In a production environment, applications span across multiple containers and these containers need to be deployed across multiple servers. With Kubernetes, it is possible to orchestrate and manage containers on hosts on a huge scale. It allows scaling and health management of container clusters.

It automatically handles logging and alerting, networking, load balancing, auto-scaling and many other tasks. The result of this is an application with good performance and low downtime requiring less to no maintenance.

Kubernetes allows the developers to move away from machines whether physical or virtual and work on a container-centric infrastructure.

It reduces the hardware cost by as much as 50% as it uses the resources efficiently.

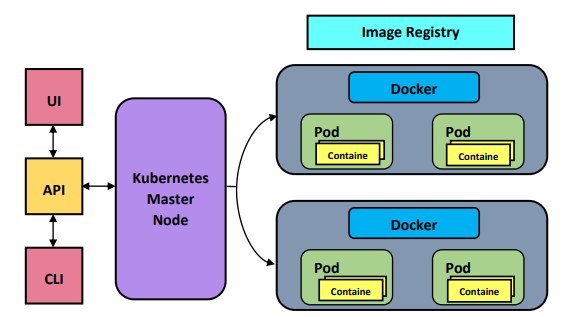
With Kubernetes, customer demands can be met efficiently and quickly:

* Deploy your applications quickly and predictably
* Scale your applications on the fly
* Roll out new features seamlessly
* Limit hardware usage to required resources only

So, to summarizeKubernetes is:

* **Portable**: public, private, hybrid, multi-cloud
* **Extensible**: modular, pluggable, hookable, composable
* **Self-healing**: auto-placement, auto-restart, auto-replication, auto-scaling

## Kubernetes Components



Kubernetes Components

**Master Node**

Master Node is the only center of cluster control and has complete control over the cluster. It is used to manage and coordinate various components of the nodes. It is also called a node controller. The Master assigns a CIDR (Classless Inter-Domain Routing) block to each node that is registered. It is responsible for interacting with cloud provider services and keeps a list of nodes along with a list of machines available on the cloud. This way, VMs can be launched as required. This is done if replication is necessary.

It also provides communication across nodes in the cluster.

It also does health checks of clusters. When a node is down, that node becomes unreachable, and it is said to be unhealthy. Health checks are done at regular intervals, and the user specifies the frequency of the health checks. If a Node has “Unknown” or “False” as its ready condition for a period longer than the “pod-evict-timeout” (the default “pod-eviction-timeout” is 5 minutes), then the Master schedules all Pods on that node for deletion.

The below JSON is returned if a node is healthy:

**Worker Node**

In Kubernetes, a node is a worker machine, and it runs the Kubernetes agent. It is also known as a minion. It can either be a physical machine or a virtual machine created by the users or a cloud provider and not by Kubernetes. Kubernetes master node manages these worker nodes.

Each node has the necessary services to manage and run pods. These services include Docker or a similar container service, kubelet and kube proxy.

**Docker**

It is necessary for every node to have a Docker component. It is used to create, modify, manage and bring the containers up or down. When a Pod is scheduled to a Node by Kubernetes, kubelet present on that Node will notify the Docker engine on the Node to launch containers. The statuses of the containers are constantly monitored and collected by kubelet from Docker. The collected information is forwarded to the Master. Based on the information, containers are started or stopped by Docker. This is done automatically in Kubernetes unlike in other tools where it must be performed manually by an administrator.

**Container**

Containers are the main components of Kubernetes. Each pod has one or more containers depending upon the application being deployed.

**Pod**

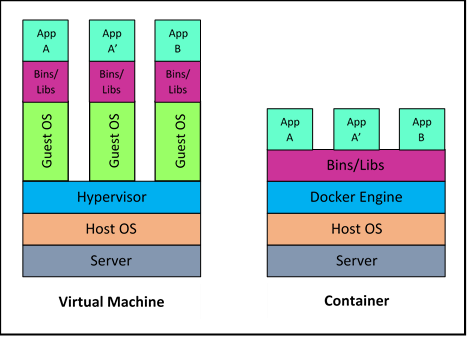
A Pod is the smallest and simplest unit among the Kubernetes object. It is a group of one or more containers that share storage and network amongst them. A pod’s contents share storage and are also scheduled parallelly.

The shared context of a pod is a set of Linux namespaces, cgroups. All containers in a pod, share IP addresses and ports, and communication between these containers is done via localhost. Containers in a Pod communicate with each other via standard inter-process communications, like POSIX shared memory or SystemV semaphores. Containers present in different Pods are assigned unique IP addresses and communication via IPC is not possible.

Applications within a pod have access to shared volumes, and these volumes can be mounted into the application’s file-system. Volumes make data persistent, i.e., data is retained after the container is restarted. It also enables data to be shared among containers within the Pod.

Pods serve as a unit of deployment, horizontal scaling, and replication. Colocation (co-scheduling), shared fate (like spinning-up and termination), coordinated replication, resource sharing, and dependency management are handled automatically for containers in a Pod.

## Why Containers



Comparison between VMs and Docker Container

Applications were deployed on a host using Operating System package managers before containers. Due to this, all the libraries, configuration, life-cycles, and executables of the applications was muddled with the host Operating System and amongst themselves. Achieving this on Virtual Machines (VM) was easier, but VMs are non-portable and heavyweight.

Now, however, containers are extensively used. Containers use Operating System-level virtualization as opposed to the hardware-level virtualization seen in the VMs. Containersisolate the application from the operating system.

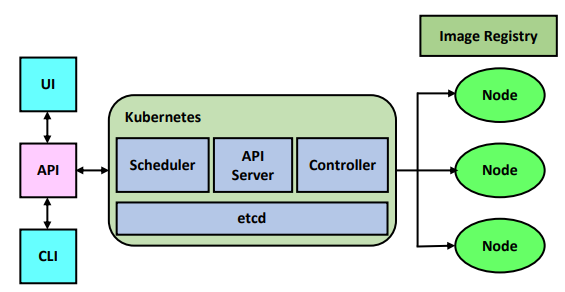
Itconsists of everything needed for an application, including all the dependencies and the stack which runs on it. This makes it extremely light-weight, scalable, and portable. It provides guaranteed execution irrespective of the underlying environment. Containers on the same host are isolated from one another. Processes are not visible to other containers, the container can restrict resources, and each can have a different file system.

# Chapter Two: Basics of Kubernetes

**Topics covered in this unit:**

* Kubernetes Master Node Architecture
* Kubernetes Worker Node Architecture
* Master-Node Communication
* Configurations
* Kubernetes Tools
* Introduction to Kops

## Kubernetes Master Node Architecture

Kubernetes Architecture

The master components provide full control over the whole Kubernetes cluster and its management. Through the master components, all the global changes of the cluster are done, along with the detection and response to the cluster events.

Master components can run on any node in the cluster, but it is recommended to run all master components on the same VM and avoid running user container on the VM, which is running master components.

**kube-APIserver**

It acts as the front-end of the Kubernetes control plane, and its primary task is to expose all the Kubernetes API. It is designed in such a way that it scales by deploying more instances, i.e., horizontal scaling.

**etcd**

All the cluster related data is stored here at etcd. It is also known as the backing store of Kubernetes. We must have a backup plan for etcd, as it is the only place all the cluster data is stored.

**kube-controller-manager**

Controllers are the background processes that are there for all the routine tasks of the cluster. kube-controller-manager runs all these controllers. All the controllers are separate processes, but they are compiled in a single binary and are run as a single process to reduce the complexity.

Following are the different controllers:

* **Node Controller**: For noticing and responding if the node is down.
* **Replication Controller**: For maintaining the right count of pods for each replication controller object in the system.
* **Endpoints Controller**: For populating the Endpoints Object.
* **Service Account and Token Controller**: For creating default accounts and API access tokens for newly created namespaces.

**Cloud Controller Manager**

Like kube-controller-manager, this also runs the controllers. But the difference is that it only runs all the controllers which are interacting with the underlying cloud providers. This binary is an alpha feature released with Kubernetes 1.6. It runs the loops based on the cloud provider, i.e., cloud-provider-specific loops, so we need to disable these lops in the kube-controller-manager. There is one more way to disable these loops by setting the flag “--cloud-provider” to “external” while starting the kube-controller-manager.

Previously, based on the cloud service provider, the core Kubernetes code used to get written. Now the as per the current release, the cloud providers themselves must maintain the provider-specific code with them and link it to the cloud-controller-manager at the runtime of Kubernetes.

**kube-scheduler**

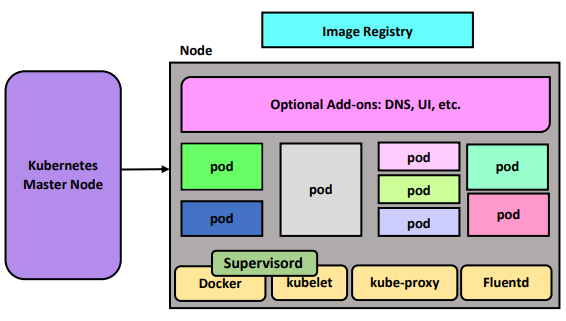
It keeps an eye on new pods that are created and don’t have any node assigned to it. Then the scheduler decides a node for the pod to run.

**Addons**

Addons are nothing but different pods that are implemented for the cluster’s features. The Deployments, ReplicationControllers, etc. can manage these addons. Namespace related addons are created in the kube-system namespace. There are various addon resources that addon manager creates and maintains:

* **DNS**: It the additional DNS server to the DNS in the environment which is dedicated to serving the DNS records for Kubernetes services. It is included by default in the containers started by Kubernetes for their DNS search.
* **Web UI**: General purpose UI for managing and troubleshooting the applications in the Kubernetes cluster. It also manages the cluster itself.
* **Container Resource Monitoring**: Provide the general metrics about the containers and gives a UI for surfing that data.
* **Cluster-level Logging**: There is a central log store that preserves all the log of the containers and provides a UI for surfing.

## Kubernetes Worker Node Architecture

****

Kubernetes Node Architecture

Node components are those which are executed on each node of the cluster for maintaining all the active pods which provide a runtime environment for Kubernetes.

Kubelet

It is a primary node agent, which keeps an eye on the pods that are assigned to the node and then carry out the following operations:

* Mounts the required volume for the pod
* Downloads the confidential data of the pod
* Runs the pod’s containers using Docker
* After a specific time, interval checks the liveliness of the container
* Conveys the pod’s status to the rest of the system
* Conveys the node’s status to the rest of the system

**Kube-proxy**

It takes care of the network rules on the host and connection forwarding, which in turn enables the abstraction in Kubernetes service.

**Docker /rkt**

Docker is used for launching containers inside the Kubernetes pods. Developers can build apps with Docker and use Kubernetes as container management tools.

Kubernetes supports Docker as container runtime and is the most commonly known runtime. Docker helps to describe pods in Docker terms.

rkt is a container engine that is used as an alternative to the Docker container engine for launching containers in the pods. It explores the alternative approach to container runtime architecture.

**Supervisord**

It keeps Kubernetes and docker running by monitoring and keeping control over the system. The primary purpose of supervisord is to create and manage processes based on data in its configuration file. Supervisord is the parent process of all the processes; it creates and manages the entirety of its lifetime.

**Fluentd**

Provides cluster-level logging which is responsible for storing container logs to a central log store and provide an interface for searching/browsing stored logs.

## Master-Node Communication

**Master to Cluster**

Master to cluster communication is mainly done in two ways: Each node has a kubelet process to which the master connects, or the master connects to any of the node objects via the master’s (apiserver) proxy.

Connection using kubelet process is mainly used for:

* Attach to running pods via kubectl
* Provide port-forwarding via kubelet
* Fetch logs on the pods

This type of master cluster connection terminates at kubelet’s end-point, and the kubelet’s service certificate is not verified by the Master leaving it susceptible to attacks.

Direct connection from Master to service, node, or a pod is simple HTTP requests and is not authenticated or encrypted. Therefore, this too is susceptible to a variety of attacks and is not recommended to be run on a public network.

To verify this connection, the kubelet-certificate-authority flag is used to provide the apiserver with a root certificate bundle which is used to verify the kubelet’s serving certificate.

**Cluster to Master**

Any cluster to master communication terminates at the apiserver on the master. Usually, the apiserver listens for HTTPS calls, and it has client authentication in place. Provisioned nodes are provided with certificates so that they can authenticate.

Kubernetes injects public root certificate and the token into the Pods so that a Pod can communicate with the master when necessary. Master communicates over an insecure port with the cluster, but the port is only exposed to the localhost. Therefore, the connection is secure and can be run on public networks as well.

## Configurations

**Configuration Best Practices**

* Latest, stable API version is to be used while defining configurations
* Having the configuration files on a version control allows for a fast roll-back in case of issues
* Although both YAML and JSON are supported, YAML is preferred as it is more user-friendly
* Grouping related objects in one file make it easier to manage
* Avoid specifying default values in configuration files, if not needed

## Kubernetes Tools

**Minikube**

Minikube is a tool used to run Kubernetes on the local machine. It sets up a single-node Kubernetes cluster running in a VM on the local machine. Since it runs on a VM, a Hypervisor is required to be installed on the host OS. VirtualBox and KVM (Linux-only) are most preferred.

It is mostly used to learn and familiarize oneself with Kubernetes and to build small PoCs.

**Kubeadm**

Kubeadm is used to create a Kubernetes cluster from the ground. It is used to set up best-practice clusters that are easy to set up and use, secure and extendable.

**kubeadm init** command is used to bootstrap a Kubernetes Master.

Security features:

* It uses RBAC (Role-based Access Control)
* Secure communication between API server and kubelets
* Secure communication between control panel components
* Restricting what Bootstrap Token can access the Master

Following a few commands are required to set up a cluster:

* **kubeadm init** command on the master
* export KUBECONFIG=/etc/kubernetes/admin.conf
* kubectl apply -f <network.yaml>
* kubeadm join –token <token> <master IP> on the node

**kubectl**

Kubectl is a command-line interface to perform various operations on Kubernetes clusters.

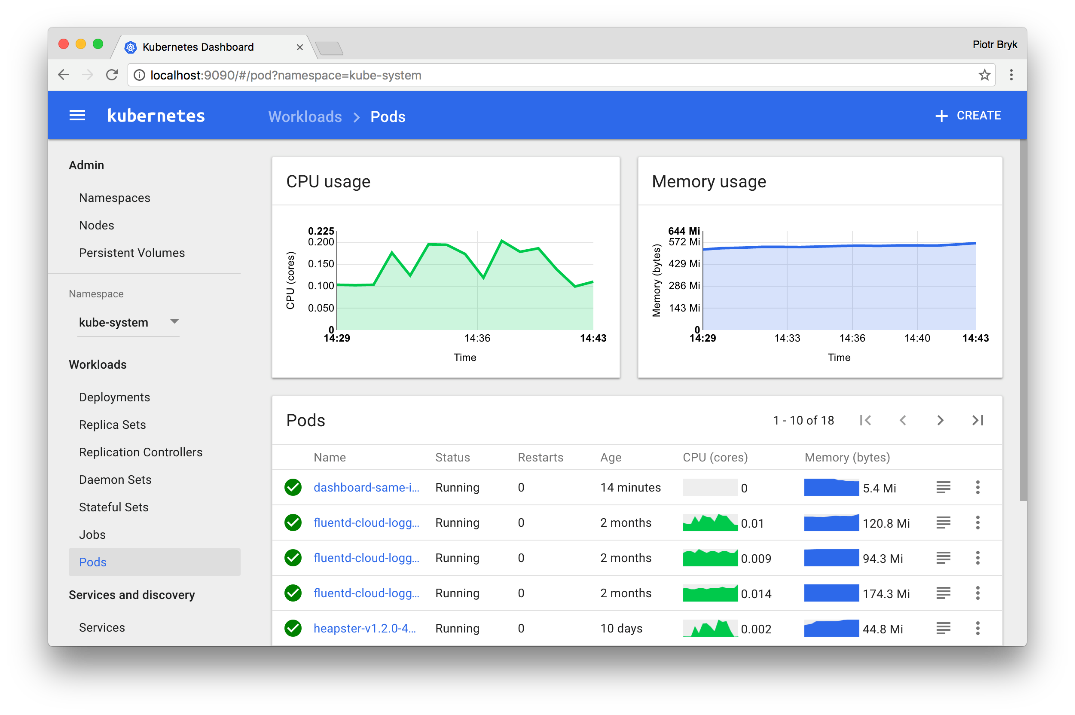
Its syntax is as follows:

kubectl [command] [TYPE] [NAME] [flags]

kubectl get pod pod0

* **command**: It states the operation that is to be performed on resources.
  + E.g., create, describe, delete, flags
* **TYPE**: It is used to specify the type of resource on which operation is to be performed. The resource type is case-sensitive and can be abbreviated.
  + E.g., pod, node
* **NAME**: It is used to specify the name of the resource on which operations are to be performed. The resource name is case-sensitive as well.

**Kubernetes Dashboard**



Kubernetes Dashboard

Kubernetes dashboard is a web-based interface for interacting with Kubernetes. It is used to deploy containerized apps to the Kubernetes cluster, troubleshoot apps, and manage the cluster.

It gives information about the usage of Kubernetes resources and provides an overview of the apps on the clusters. It is also used to manage and modify Kubernetes resources.

## Introduction to kops

Kops is a part of the Kubernetes project, and it is used to manage production-grade Kubernetes clusters. It is popularly used for Kubernetes cluster deployment on Amazon Web Services, and it automates a large part of Kubernetes operations on Amazon Web Services.

It does cluster creation and cluster configuration using a declarative model. This is a model in which the desired result is fed, and the tool is made such that it can figure out what steps are to be taken to achieve the desired result.

Key features of Kops:

* Deploying Kubernetes clusters on VPC
* Allows for provisioning of one or more master clusters
* Supports heterogeneous clusters by creating multiple instance groups
* Rolling updates for clusters
* Works with infrastructure as code tools like CloudFormation or Terraform.

# Lab 1: Kubernetes Operations on AWS

A Kubernetes cluster is created on AWS using the Kubernetes Operations(kops) tool.

Topics

* Launching an EC2 instance
* Setting up a Kubernetes cluster using Kubernetes Operations
* Creating a cluster with/without DNS

Before proceeding, obtain a set of API keys for your user and create a keypair in EC2.

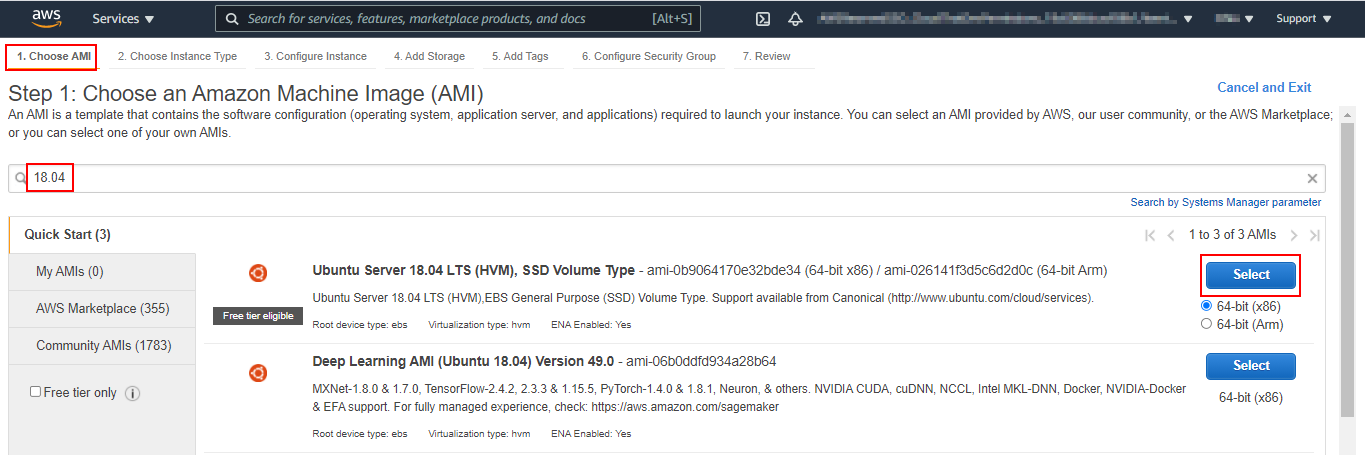
## Task 1: Launching an EC2 Instance

1. Navigate to **EC2** in the AWS console and select **Launch Instance**

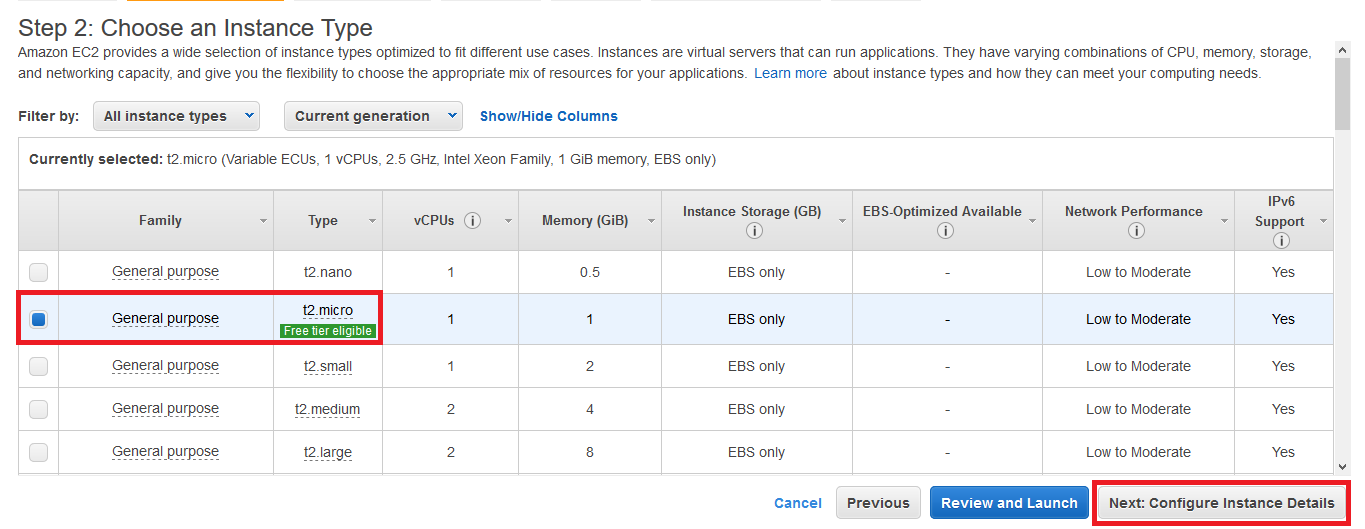
**Graphical user interface, website

Description automatically generated**

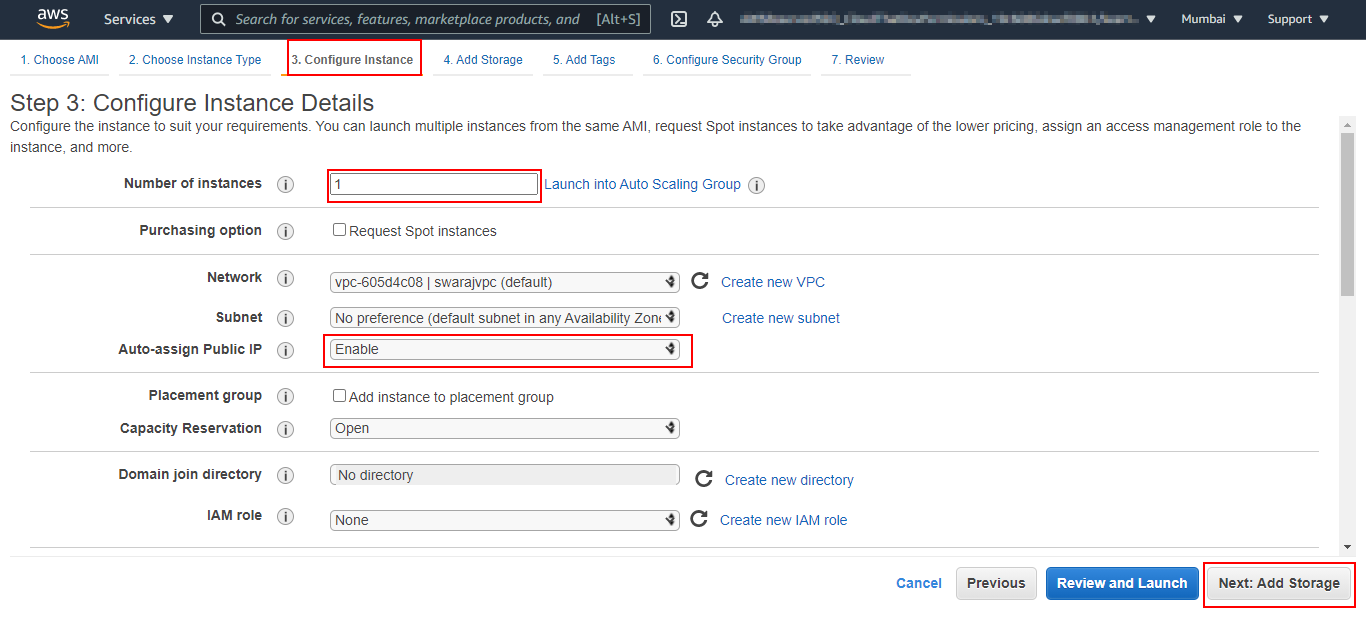
1. Select **Ubuntu Server 18.04 LTS** AMI



1. Select **t2.micro** as the **Instance Type** select **Next: Configure Instance Details**



1. Make sure **Auto-assign Public IP** is set to **Enable** and select **Next: Add Storage**

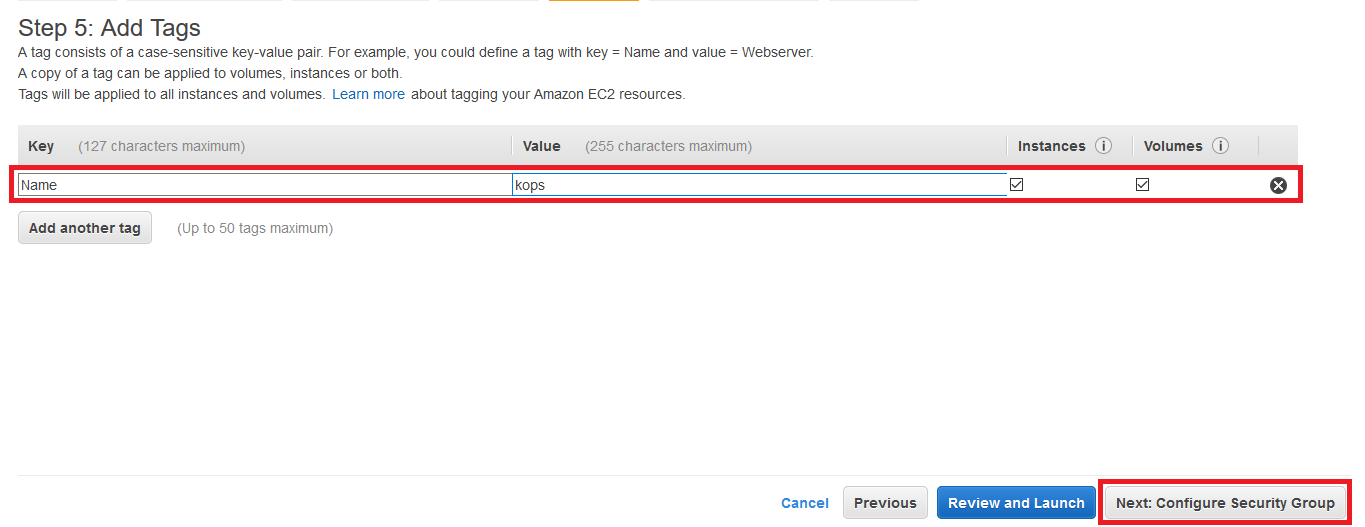


1. Change Volume **Size** to **10** (GiB) and select **Next: Add Tags**

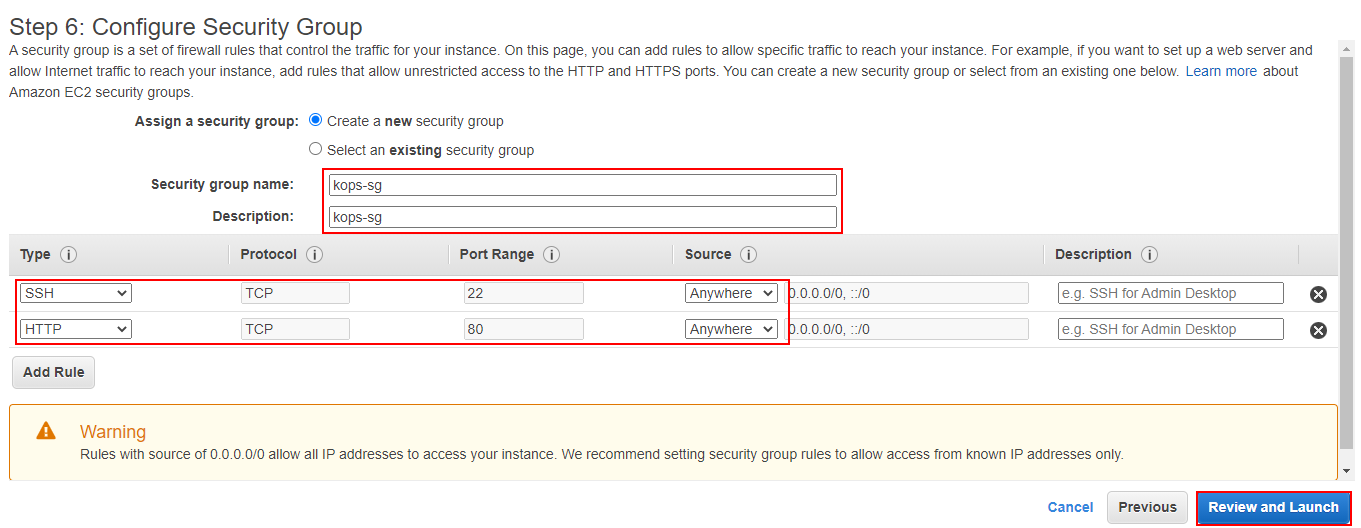
A screenshot of a computer

Description automatically generated

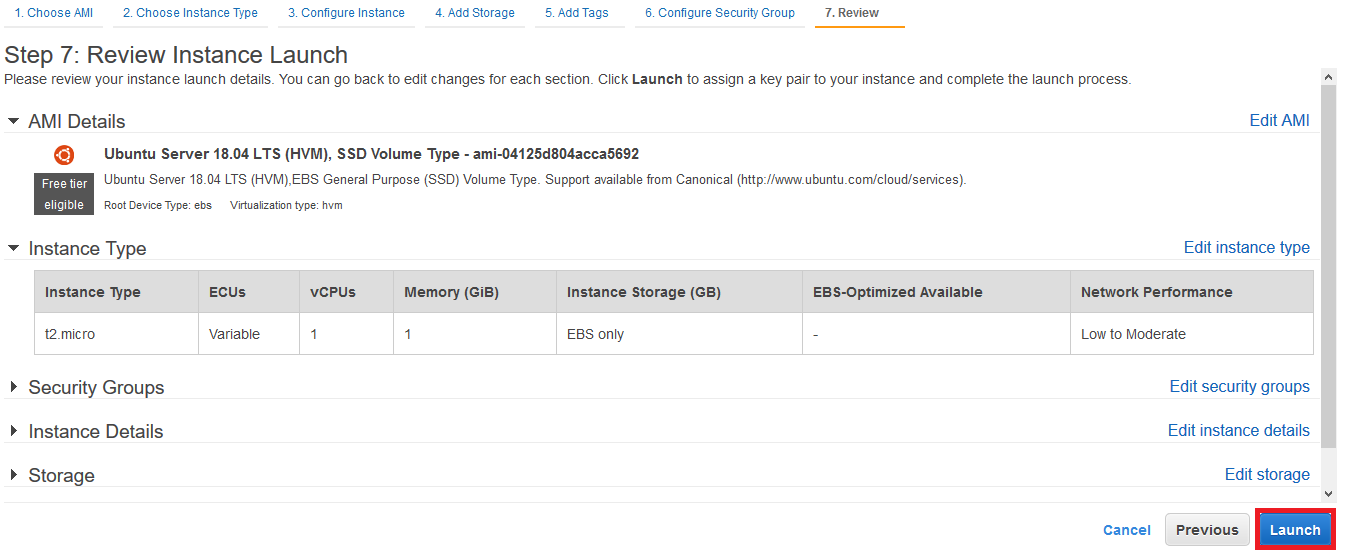
1. Add **Name** under **Key** and **kops** under **Value.** Then click **Next: Configure Security Group**



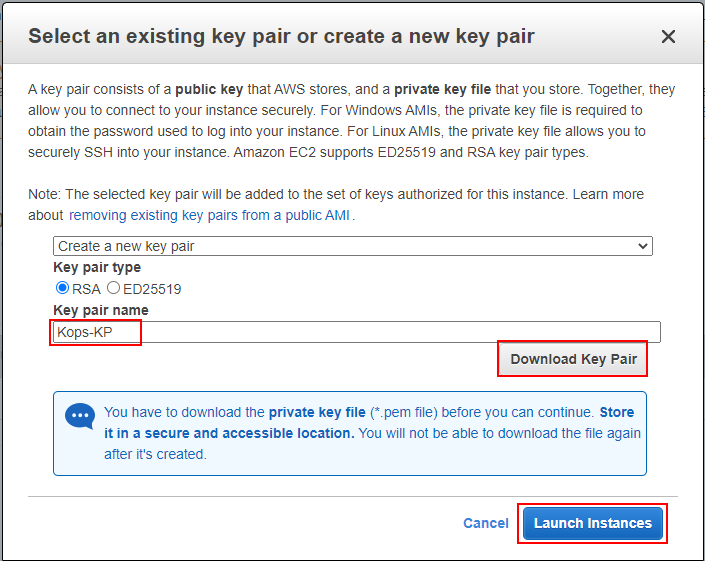
1. Select **Create a new Security Group** and name it as **kops-sg.** Change **Source** to **Anywhere** for **SSH and HTTP.** Thenclick **Review and Launch**



1. Click **Launch** on the next page

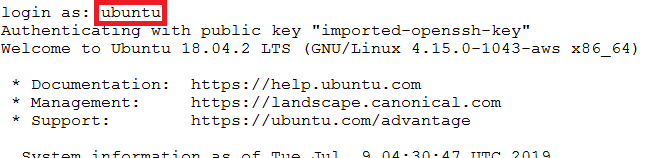


1. Create a new keypair named **Kops-KP** download the keypair by clicking **Download Key Pair** option and then click **Launch Instances**



## Task 2: Setting up a Kubernetes Cluster

1. SSH into the machine using the keypair and the public IP and log in with user **ubuntu**



1. Download the script to setup KOPS cluster using the link

$ wget files.cloudthat.training/devops/kubernetes-essentials/kc-install-aws-kops-kv1.17.sh

Text, email

Description automatically generated

1. Run the script to setup and configure the Kubernetes cluster. Enter the AWS keys, Availability Zone, and cluster name when prompted. Cluster name needs to be in the format **<name>.k8s.local**

$ bash kc-install-aws-kops-kv1.17.sh

****

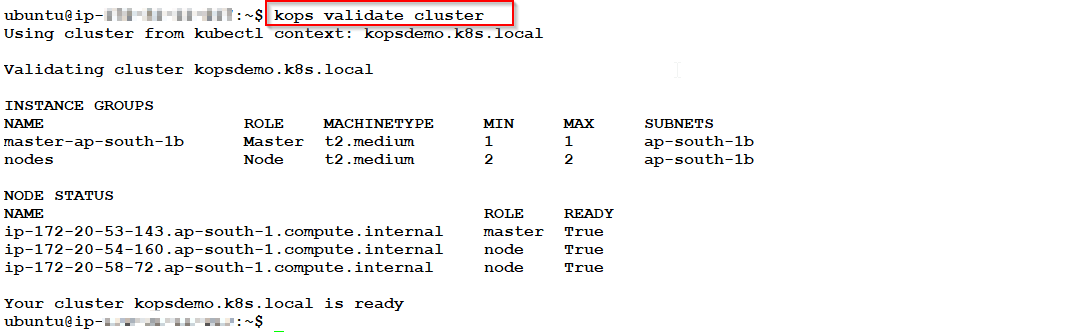
1. Set S3 bucket environment variable. Bucket name will be same as the name of cluster entered while executing a script

$ export KOPS\_STATE\_STORE=s3://< Cluster\_Name >



1. Validate cluster creation

$ kops validate cluster



**Note**: It may take up to 15 minutes for the cluster to be fully up.

1. Set the hostname to kops by running below command

$ sudo hostnamectl set-hostname kops

**Note**: Please take a new putty session to reflect the changed hostname.

# Chapter Three: Kubernetes Administration

**Topics covered in this unit:**

* Administration of Kubernetes Cluster
* Kubernetes Objects
* Pod
* Services
* Storage
* Load Balancing

## Administer a Cluster

Administrating a cluster is not a single task; it is like a seamless collaboration of multiple administrative tasks that are required to manage the cluster at its best. The cluster administration starts from the very beginning.

It will include the following steps:

* Planning the cluster
* Managing the cluster
* Securing the cluster

**Planning the Cluster**

For planning the cluster, we need to take some points into considerations, based on which we can decide which solution to implement. Following are the points that we need to check for:

* For what will we be using Kubernetes cluster? Either for POC or a highly available multi-node solution
* If we are planning to implement highly available multi-node cluster, then need to get very clear with the multi-zone cluster configuration
* Willing to have our own cluster or a hosted cluster on Google Container Engine
* It will be an on-premise or on the cloud. If the requirement is for the hybrid cluster, we need to set up multiple clusters as Kubernetes does not support hybrid cluster
* For on-premise setup, make sure that you are selecting the most suitable networking model
* The cluster will be on bare metal hardware or virtual machines

**Managing Cluster**

Managing a cluster includes multiple small and big parts which play a vital role, individually and in collaboration as well. Managing a cluster begins from creating a cluster, upgrading both master and the worker nodes, maintaining nodes until updating the API version of the running cluster.

For managing the cluster properly, node status plays an important role in providing all the required information about the nodes in the cluster. Such as hostnames, the status of all running nodes, which resources are available on the node, and other general information about the node such as versions of Kernel, Kubernetes, Docker, the name of the OS.

**Namespaces**

Kubernetes supports multiple virtual clusters called Namespaces on the same physical cluster. Namespaces are intended to be used in an environment with many users across multiple projects.

Namespaces are used to divide cluster resources between users. Labels are used to differentiate resources in a namespace.

**Labels**

apiVersion: v1 Version of Kubernetes API to create this object

kind: Pod This defines the object type

metadata: Data that helps to uniquely identify the object

name: nginx

labels:

name: nginx

spec: spec part of the Object

containers:

- name: nginx

image: nginx

ports:

- containerPort: 80

Labels are key-value pairs attached to Kubernetes objects like containers. They are used to assign identifying parameters to objects that are easier for a user to understand.

They are attached to objects at the time of creation and can be modified later.

Each object has a unique key-value pair attached to it.

**Selectors**

Selectors also called label selectors are used to providing uniqueness to labels as multiple objects can have the same label. It is used to identify a set of objects.

The Kubernetes API currently supports two types of selectors:

* Equality-based: Performs filtering by a key-value pair.
* Set-based: Filters keys by a set/group of values.

## Kubernetes Objects

The persistent entities in the Kubernetes system are known as Kubernetes Objects.

These objects represent the state of the cluster. Specifically, these objects describe:

* What containerized applications are running (and on which nodes)
* The resources available to those applications
* The policies for applications behavior, such as restart policies, upgrades, and fault-tolerance

A Kubernetes object describes how the cluster workload should look. It defines the **desired state** of the cluster. Once the object is created, the Kubernetes system constantly tries to ensure that the object is existing.

To create, modify, or delete a Kubernetes object, we need to use Kubernetes API. When we use Kubectl command-line interface, it internally makes Kubernetes API calls. Kubernetes API calls can also be made directly from our own programs.

Kubernetes Objects have two fields which define its configuration:

* Object spec: spec defines the desired state of the object and some basic information.
* Object status: status defines the actual state of the object.

**Pods**

Pods are the smallest components of Kubernetes, and they host one or more containers.

**Pod overview**

The Pod is the most fundamental part of Kubernetes. Each pod is a process that is running on the cluster. It is a group of one or more containers.

A Pod encloses an application container, unique IP address, and storage. It consists of one or more containers running an instance of the application. These containers share resources amongst them and are tightly coupled.

Pods are made to manage multiple containers, and they are co-located and co-scheduled on the same host within the cluster. These containers share resources and dependencies.

Although Docker is mostly used in Pods, Kubernetes Pods can house other containers as well.

Pods are run in two ways: A Single container in each Pod and multiple containers in each Pod.

One container in each Pod is the most popular way. Pod acts as a wrapper for a container. Here Kubernetes indirectly manages the container via its Pod.

Multiple containers per Pod is mostly used when containers all contribute in a unique way to accomplish a collective goal. Grouping them this way, makes management tasks easier.

The controller node can create and manage Pods. It also provides automatic replication and self-healing.

**Uses of Pods:**

* Log trails, data loggers
* Network proxies, adaptors, and bridges
* Log backup, checkpoint backup, compression and snapshots
* Content management systems

The following block shows an example YAML used to launch an NGINX pod:

apiVersion: v1

kind: Pod

metadata:

name: nginx

labels:

name: nginx

spec:

containers:

- name: nginx

image: nginx

ports:

- containerPort: 80

## Pod Lifecycle

**Pod Phase**

Pod phase gives a top-level summary of where a Pod is in its lifecycle. A Pod’s status is defined by Phase.

Listed below are some of the values that Phase takes:

* **Pending:** Pod is registered in Kubernetes, but the container images are being created.
* **Running:** Pod has been defined on a node, all containers are created, and at least one of them is running.
* **Succeeded:** Containers in the pod have finished their task and have terminated successfully.
* **Failed:** At least one container in the Pod has failed, and all the containers have terminated.
* **Unknown:** Status of the Pod cannot be determined due to an error in communication.

**Container Probe**

Container probe is a diagnostic performed from time to time on containers by kubelet. It runs three diagnostic tests: runs the command in the container, TCP check on its IP on a specific port, and an HTTP get request.

The outcome can be Success, Failure, or Unknown.

**Restart Policy**

The restart policy takes three values: Always, OnFailure, and Never. It is set to Always by default. This policy applies to all the containers in the Pod. Containers that have Failed and need to be restarted by Kubelet are restarted with an exponential back-off delay ending at 5 minutes.

**Pod Lifetime**

Pods are not removed until it is explicitly removed. The admin or the controller can perform this action. But pods having either Succeeded or Failed Phase for over a period (set in the Master), are destroyed automatically.

Kubernetes has three types of controllers:

* Uses jobs for Pods that are assumed to terminate. Jobs run Pods to completion. Therefore it can only be used with a restart policy of Never or OnFailure
* Uses ReplicationController, ReplicaSet, or Deployment on Pods that are not expected to terminate. Therefore, it is used only on Pods having a restart policy of Always
* Uses DaemonSet on Pods that run one per machine

If any node disconnects, all the Pods on it are set to Failed Phase by Kubernetes.

**Init Containers**

Init containers are containers that are started before starting the app containers in Pod. The only differences are that Init containers run to completion each time and every init container makes sure the previous one has reached completion before starting.

If an Init container fails, the Pod with the Container is restarted until the Container(s) reach completion.

Only when all the Init containers are run successfully, the Pod is initialized, and all the app containers start.

Init containers are mainly used for:

* Running utilities as it’s not suitable to be executed on the app container
* Custom code which is not present in the app
* Any task requiring greater security as it uses Linux namespaces and it has access to certain objects app containers do not have
* Setting a desirable environment for the app containers before they start

**Disruptions**

Pods do not get destroyed automatically unless its setup by the controller or a user. It is sometimes automatically removed if there is a fatal hardware or software error.

Such unavoidable and unforeseen errors are involuntary disruptions. Such disruptions can be caused by:

* A hardware failure on the physical machine
* Accidental deletion of the VM
* Kernel panics
* Removal of the node running out of resources
* Hypervisor failure on a cloud platform

Voluntary disruptions are ones triggered by the administrator or the app owner. Such disruptions are caused by:

* Removing the deployment or a controller managing the Pod
* Updating the deployment template
* Accidentally deleting a Pod
* Removing resources allocated to a node
* Manual scaling of nodes

**Dealing with Disruptions**

* Pods must be made to request the resource needed by it to run the apps to completion
* Replication of the apps for high-availability
* Spanning apps across racks or multiple availability zones

Unless the cluster administrator or the cloud provider is constantly making changes, voluntary disruptions are not possible. If the frequency of voluntary disruptions can be estimated, a high availability setup can be made possible. This is provided by a Kubernetes feature called Disruption Budgets.

It will also be beneficial to separate the roles of the app owner and the cluster administrator while having limited knowledge of each other. The Pod Disruption Budgets support separation, and it provides an interface between the roles.

Performing disruptive action on Nodes:

* Failover to replica clusters
* Apps made disruption-friendly
* Using Pod Disruption Budgets

## Services

Kubernetes Service groups a logical set of Pods and defines the policies to access them. A Label Selector determines the set of pods that are grouped. For example, consider an image processing application it has a frontend part running on one Pod which is accessing 3 Pods in the backend which are replicas of each other. While the Pods running in the backend might die and get resurrected by the replication controller, the frontend application need not keep track of the backend Pods IPs. The Service abstraction enables this decoupling.

**Defining a Service**

A service is a REST object, and like other REST objects it’s definition can be POSTed to the API-server to create an instance. Following is the yaml file for creating a service that groups a logical set of Pods carrying a label “app=MyApp”. They all expose port 9376.

kind: Service

apiVersion: v1

metadata:

name: my-service

spec:

selector:

app: MyApp

ports:

- protocol: TCP

port: 80

targetPort: 9376

**Multi-Port Services:** Services in Kubernetes can expose more than one. Service objects can have multiple port definitions. When using multiple ports, you must give all your ports names, so that endpoints can be disambiguated. For example:

spec:

selector:

app: MyApp

kind: Service

apiVersion: v1

metadata:

name: my-service

ports:

- name: http

protocol: TCP

port: 80

targetPort: 9376

- name: https

protocol: TCP

port: 443

targetPort: 9377

**Headless Service**

Sometimes we don’t need or want load-balancing and a single service IP. In this case, you can create “headless” services by specifying "None" for the cluster IP (spec.clusterIP).

This alternative enables developers to decrease coupling to the Kubernetes system by enabling them the opportunity to discover their specific way. Applications can still use a self-registration pattern and adapters for other discovery systems could easily be built upon this API.

For such Services, a cluster IP is not allocated, kube-proxy does not handle these services, and there is no load balancing or proxying done by the platform for them. How DNS is automatically configured depends on whether the service has selectors defined.

A cluster IP doesn’t get allocated for those services as the kube-proxy does not handle these services. Also, for these services, no proxying or load-balancing is done.

**With Selectors**

For headless services that define selectors, the endpoints controller creates Endpoints records in the API and modifies the DNS configuration to return A records (addresses) that point directly to the Pods backing the Service.

**Without Selectors**

For headless services that do not define selectors, the endpoints controller does not create Endpoint records. However, the DNS system looks and configures either:

* CNAME records for ExternalName-type services
* A record for any Endpoints that share a name with the service, for all other types.

**Publishing Services - Service Types**

For some parts of the application (e.g., frontends), you may want to expose a Service onto an external (outside of your cluster) IP address.

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.

**Service Discovery**

Pods in the Kubernetes clusters are ephemeral, and therefore, IP keeps on changing as they come and go. For accessing the pods through the service, it is required that the Service dynamically searches and identifies the pods that are implementing the Service. This is called Service Discovery.

Pods are ephemeral. In case of any failures, the replication controller can reschedule the pods on different nodes. Pods are discovered by default by the Kubernetes service by the DNS names.

Environment variables and DNS are the two primary modes of finding a service.

* **Environment Variables**: A set of environment variables is added by kubelet for each active Service when a Pod starts running on a Node.
* Variables {SVCNAME}\_SERVICE\_HOST and {SVCNAME}\_SERVICE\_PORT which are docker compatible are supported by the Kubernetes.
* **DNS**: An optional (however strongly prescribed) cluster add-on is a DNS server. The DNS server watches the Kubernetes API for new Services and arranges of DNS records for each. If DNS has been empowered throughout the cluster, at that point, all Pods ought to have the capacity to make name resolution of Services consequently. For instance, on the off chance that you have a Service called "my-service" in Kubernetes Namespace "my-ns" a DNS record for "my-service.my-ns" is made. Cases that exist in the "my-ns" Namespace ought to have the capacity to discover it by basically doing a name lookup for "my-service". Units that exist in different Namespaces must qualify the name as "my-service.my-ns". The aftereffect of these name lookups is the cluster IP. Kubernetes additionally bolsters DNS SRV (service) records for named ports. On the off chance that the "my-service.my-ns" Service has a port named "http" with convention TCP; you can do a DNS SRV query for "\_http.\_tcp.my-service.my-ns" to find the port number for "http".

## Load Balancing

There are two types of Load balancing in Kubernetes:

* **Internal**

Services achieve internal Load Balancing in Kubernetes. Every pod exposes ClusterIP and pods exposed by the services in the cluster as an environment variable.

* **External**

NodePort exposes a high-level port for every specified node in the cluster, externally. When the number of Pods exceeds 100 or more on the Nodes, this becomes less stellar. So, there is a need for the external load balancer to do the port translation which isn’t optimal.

The solution to the problem mentioned above is an external load balancer if the user is running the cluster on GCE or any other cloud vendor. The pods get exposed to a high range external port, and the load balancer routes directly to the pods.

## Storage

**Volumes**

Previously the files on the disk were facing two major issues. First, if the container is crashing, the kubelet will restart the container, but it will start in a clean state, i.e., no files will persist there. Secondly, if we are running more than one container together in a Pod, we need to share the files between those containers.

The volume abstraction feature of Kubernetes solves these issues. These volumes have an overt lifetime as like the Pods. Therefore, the volume remains in the container that is running inside the Pod, and the whole data persists even after the restart of the container.

Kubernetes supports a wide variety of volumes, and Pod can use multiple types of volumes at the same time.

**Example:**

apiVersion: v1

kind: Pod

metadata:

name: redis

spec:

containers:

- name: redis

image: redis

volumeMounts:

- name: redis-storage

mountPath: /data/redis

volumes:

- name: redis-storage

emptyDir: {}

**Supported Types**

|  |  |
| --- | --- |
| * emptyDir * hostPath * gcePersistentDisk * awsElasticBlockStore * nfs * iscsi * fc (fibre channel) * flocker * glusterfs * rbd * cephfs * gitRepo | * secret * PersistentVolumeClaim * downwardAPI * projected * azureFileVolume * azureDisk * vsphereVolume * Quobyte * PortworxVolume * Scaleio * StorageOS * Local |

**Persistent Volume**

A Persistent Volume or PV is used for storage and is a part of the cluster which is created by an administrator. It is another resource in the cluster, just like a Node or a Pod. The lifecycle of PVs is independent of the Pods that make use of them.

Below yaml shows a PersistentVolume of 20GiB with a ReadWriteMany (RWX) access mode. It is of hostPath type.

apiVersion: v1

**kind: PersistentVolume**

metadata:

name: pv-volume

labels:

type: local

spec:

storageClassName: manual

capacity:

**storage: 20Gi**

accessModes:

**- ReadWriteMany**

**hostPath:**

path: "/pv/data"

**Persistent Volume Claim**

A Persistent Volume Claim or a PVC is a user request to allocate a Persistent Volume. Persistent Volume Claim consumes Persistent Volume resources. A request consists of storage size and mode of access. Access modes are of three types:

* **ReadWriteOnce (RWO)**: volume is mounted as read-write for a single node
* **ReadOnlyMany (ROX)**: volume is mounted read-only for many nodes
* **ReadWriteMany (RWX)**: volume is mounted as read-write for many nodes

A PersistentVolume is provisioned in two ways, Static and Dynamic:

**Static**

PersistentVolumes are created by an administrator and exist in Kubernetes API available for consumption

**Dynamic**

If no statically provisioned PersistentVolume matches a PersistentVolumeClaim made by a user, the cluster may try to provision a volume for the PersistentVolumeClaim dynamically.

Dynamic provisioning is based on StorageClasses. Storage class must be created and configured in advance by a cluster administrator for dynamic provisioning.

Below yaml shows a PersistentVolumeClaim request of 10GiB with a ReadWriteMany (RWX) access mode.

apiVersion: v1

**kind**: **PersistentVolumeClaim**

metadata:

name: pv-claim

spec:

storageClassName: manual

accessModes:

- **ReadWriteMany**

resources:

requests:

**storage**: **10Gi**

**StorageClass**

StorageClass is used by cluster administrators to describe the class of storage that is provisioned. Every StorageClass has properties of provisioner, parameters, and reclaim policy.

Provisioner **determines which plugin was used to provision the PersistentVolume. Parameters** of a StorageClass varies based on its provisioner. **ReclaimPolicy** can either be Retain or Delete. The default value of reclaimPolicy is **Delete**.

**Binding**

A PersistentVolumeClaim has already been made with some storage size and mode of access. Kubernetes Master consists of a control loop that constantly monitors for new PersistentVolumeClaim. Once a PersistentVolumeClaim is made, the control loop binds the claim with a PeristentVolume.

**Reclaim**

Once the user is finished running tasks on a volume, PVC objects from the API can be deleted allowing for the resource to be reclaimed. PersistentVolume **reclaimPolicy** informs the cluster what to do with the volume once it’s released from its claim. Volumes can be **Deleted, Retained, or Recycled.**

* **Delete**: Removes the PersistentVolume object from Kubernetes along with its associated storage in the infrastructure
* **Retain**: Allows for manual reclamation of the resource. Once a PersistentVolumeClaim is deleted, the PersistentVolume associated with it remains and is considered to be **released.** As the data from the previous claim is left behind on the volume, it is not available for another claim. This data must be cleaned up manually.
* **Recycle**: Allows for reclamation of a resource similar to Retain. Once a PersistentVolumeClaim is deleted, it erases the data from the previous claim and makes the volume available for another claim.

## Stateless Applications

* Stateless applications don’t keep track of previous transactions. Each interaction request is handled based entirely on information that comes along with the request. Each interaction is handled These applications do not manage its state in the Kubernetes cluster as well
* A stateless Kubernetes application is an application that doesn't manage its state in the Kubernetes cluster. All the states are stored outside the cluster, and the cluster containers access it in some manner
* Deployments and ReplicationControllers in Kubernetes are meant to be used with Stateless Applications

## Stateful Applications

* A stateful application keeps track of all the changes its undergone since it started running. It is a common misconception that containers only run stateless applications. Many container orchestration systems including Kubernetes can handle both stateful and stateless applications
* Statefulset in Kubernetes is used for of Stateful applications or Stateful storage.

# Lab 2: Services in Kubernetes

A httpd pod is created and service of type **LoadBalancer** is made for the Pod. This service creates a load balancer on AWS which allows for external access.

Topics

* Create a pod using yaml
* Setup ClusterIP service
* Setup NodePort service
* Setup LoadBalancer service
* Delete and recreate httpd pod

## Task 1: Create a pod using yaml

1. Create a httpd pod yaml file with the content below

$ vi httpd-pod.yaml

Shape

Description automatically generated

apiVersion: v1

kind: Pod

metadata:

name: httpd-pod

labels:

app: httpd-ws

spec:

containers:

- name: httpd-container

image: httpd

ports:

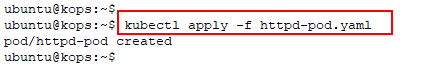
- containerPort: 80

Once entered, the file should look as shown below



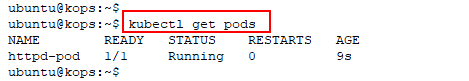
1. Apply the httpd pod file.

$ kubectl apply -f httpd-pod.yaml



1. View the newly created pod

$ kubectl get pods



## Task 2: Setup ClusterIP service

1. Create a service using the service yaml with the content given below

$ vi httpd-svc.yaml



apiVersion: v1

kind: Service

metadata:

name: httpd-svc

spec:

***selector:***

***app: httpd-ws***

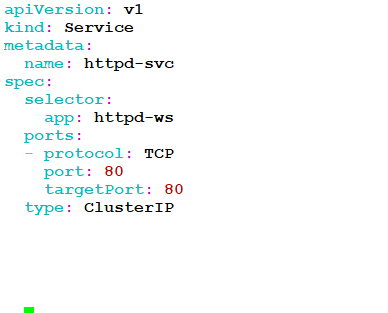
ports:

- protocol: TCP

port: 80

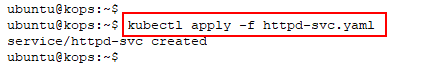
targetPort: 80

***type: ClusterIP***



1. Apply the service yaml

$ kubectl apply -f httpd-svc.yaml



1. Describe the service created and notice that it has **endpoints**

$ kubectl describe svc httpd-svc

Text

Description automatically generated

1. Try to access the service from the **kops machine** using the Cluster IP and see that it is not accessible as it is of the type **ClusterIP**

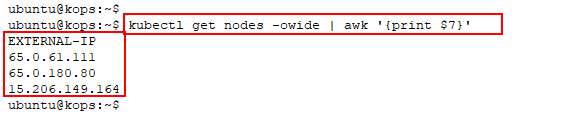
$ curl <Cluster\_IP>:<Port\_Number>

Graphical user interface, application

Description automatically generated

1. Get External IPs of the machines in the cluster. SSH to one of the machines and rerun the command in the previous task

$ kubectl get nodes -owide | awk '{print $7}'



$ ssh -t admin@<Node\_IP> curl <Cluster\_IP>:<Service\_Port>

Graphical user interface, text

Description automatically generated with medium confidence

## Task 3: Setup NodePort Service

1. Modify the service created in the previous task to type **NodePort**. Open httpd-svc.yaml file using an editor and replace the **ClusterIP** to **NodePort**

$ vi httpd-svc.yaml

Shape

Description automatically generated

apiVersion: v1

kind: Service

metadata:

name: httpd-svc

spec:

***selector:***

***app: httpd-ws***

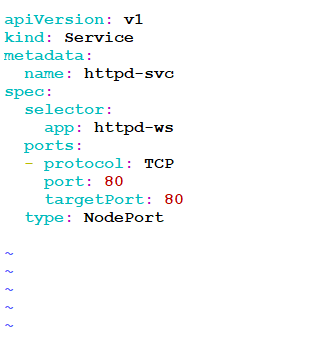
ports:

- protocol: TCP

port: 80

targetPort: 80

***type: NodePort***



1. Apply the changes using below command

$ kubectl apply -f httpd-svc.yaml

A picture containing text

Description automatically generated

1. View details of the modified service

$ kubectl describe svc httpd-svc

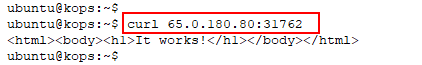
Graphical user interface, text, application, email

Description automatically generated

1. Access the service from the kops instance, **which is** **not a part of the cluster**. The service will provide external access using NodePort on the machine’s EXTERNAL-IP. Remember to add security group rules in worker node to allow traffic for the port range of 30000 – 32767

**Note**: The **NodePort (30000 – 32767)** has to be opened in the AWS Security group of the instance that is being accessed

$ curl <EXTERNAL-IP>:NodePort



## Task 4: Setup LoadBalancer service

1. Open httpd-svc.html file using an editor and replace **NodePort** to **LoadBalancer**

$ vi httpd-svc.yaml

Shape

Description automatically generated

apiVersion: v1

kind: Service

metadata:

name: httpd-svc

spec:

***selector:***

***app: httpd-ws***

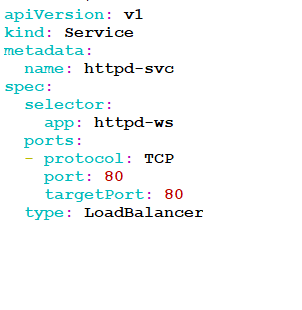
ports:

- protocol: TCP

port: 80

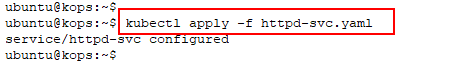
targetPort: 80

***type: LoadBalancer***



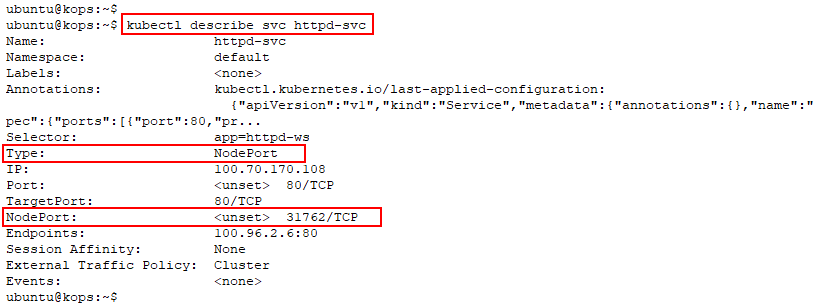
1. Apply the changes with below command

$ kubectl apply -f httpd-svc.yaml



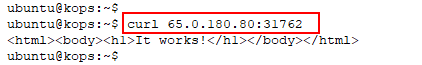
1. View details of the modified service

$ kubectl describe svc httpd-svc



1. Access the **LoadBalancer** **Ingress** on the kops instance

$ curl <LoadBalancer\_Ingress>:<Port\_number>

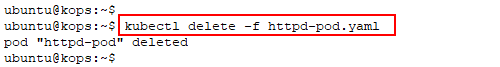


Note: It may take few seconds to Loadbalancer up and running

## Task 5: Delete and recreate httpd Pod

1. Delete the httpd Pod created in the previous task

$ kubectl delete -f httpd-pod.yaml



1. View the service details and notice that the **Endpoints** field is empty

$ kubectl describe svc httpd-svc

Graphical user interface, text

Description automatically generated with medium confidence

1. Recreate the httpd Pod and view service details. See that the **Endpoints** field is now populated

$ kubectl apply -f httpd-pod.yaml

$ kubectl describe svc httpd-svc

A picture containing logo

Description automatically generated

Graphical user interface, text, application

Description automatically generated

## Task 6: Clean-up

1. Clean up all the resources created in this task.

$ kubectl delete -f httpd-pod.yaml

$ kubectl delete -f httpd-svc.yaml

A picture containing diagram

Description automatically generated

A picture containing graphical user interface

Description automatically generated

# Lab 3: Persistent Volume in Kubernetes

This lab demonstrates the functionality and application of Persistent Volumes in Kubernetes.

**Topics**

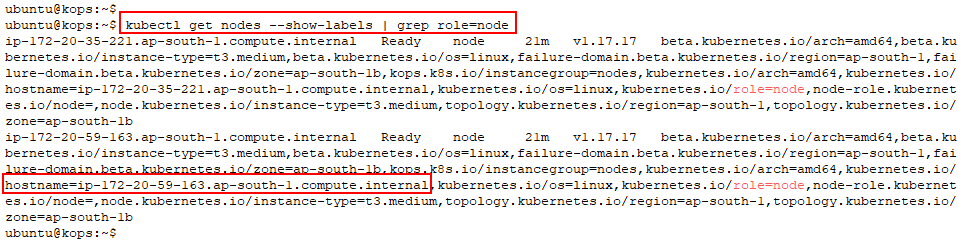
* Get node label and create custom index.html on node
* Create a local PersistentVolume
* Create a PersistentVolumeClaim
* Create nginx Pod with nodeSelector

## Task 1: Get Node Label and Create Custom Index.html on Node

1. View nodes and their labels

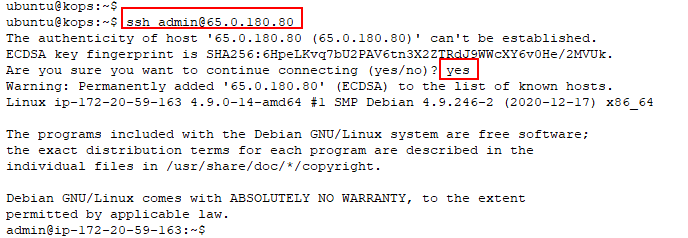
Make a note of the **kubernetes.io/hostname** label of **one** of the nodes

$ kubectl get nodes --show-labels | grep role=node



1. SSH into one of the nodes using its **public IP**. Make sure it is the same node of which the hostname is noted

$ ssh admin@<node\_public\_IP>



1. Switch to root and run the following commands. A directory with custom **index.html** is created for PersistentVolume mount

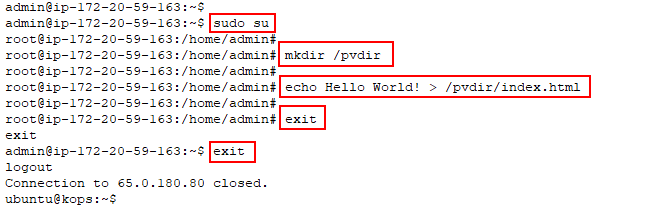
$ sudo su

# mkdir /pvdir

# echo Hello World! > /pvdir/index.html

# exit

# exit

****

## Task 2: Create a Local Persistent Volume

1. Create a yaml called **pv-volume.yaml** and enter its contents as given below

$ vi pv-volume.yaml



kind: PersistentVolume

apiVersion: v1

metadata:

name: pv-volume

labels:

type: local

spec:

storageClassName: manual

capacity:

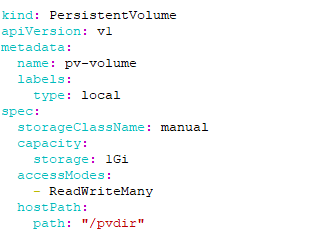
storage: 1Gi

accessModes:

- ReadWriteMany

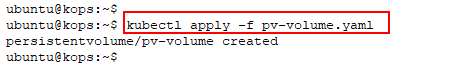
hostPath:

path: "/pvdir"

****

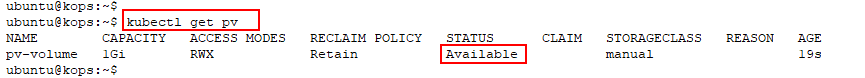
1. Apply the PersistentVolume yaml created in the previous step

$ kubectl apply -f pv-volume.yaml



1. View details of the PersistentVolume

$ kubectl get pv



## Task 3: Create a PersistentVolumeClaim

1. Create a yaml called **pv-claim.yaml** and enter its contents as given below

$ vi pv-claim.yaml

Shape

Description automatically generated with low confidence

kind: PersistentVolumeClaim

apiVersion: v1

metadata:

name: pv-claim

spec:

storageClassName: manual

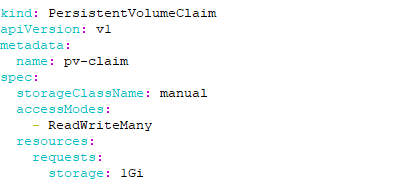
accessModes:

- ReadWriteMany

resources:

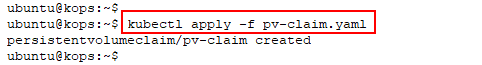
requests:

storage: 1Gi



1. Apply the PersistentVolumeClaim yaml created in the previous step

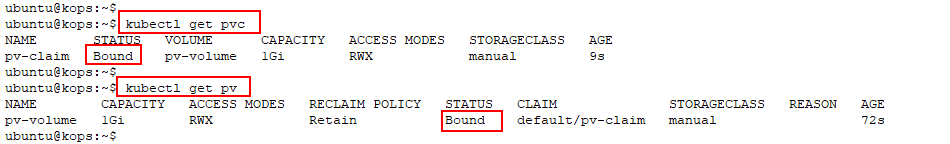
$ kubectl apply -f pv-claim.yaml



1. View details of the PersistentVolumeClaim. Ensure its status is **Bound**

$ kubectl get pvc

$ kubectl get pv



## Task 4: Create nginx Pod with NodeSelector

1. Create a yaml called **pv-pod.yaml** and enter its contents as given below. NodeSelector value is entered from hostname value noted from Task 1

$ vi pv-pod.yaml



kind: Pod

apiVersion: v1

metadata:

name: pv-pod

spec:

volumes:

- name: pv-storage

persistentVolumeClaim:

claimName: pv-claim

containers:

- name: pv-container

image: nginx

ports:

- containerPort: 80

name: "http-server"

volumeMounts:

- mountPath: "/usr/share/nginx/html"

name: pv-storage

nodeSelector:

kubernetes.io/hostname: **<Your Node Hostname>**

**Note**: The “**:**” has to be used to map and not “=” in the nodeSelector hostname.



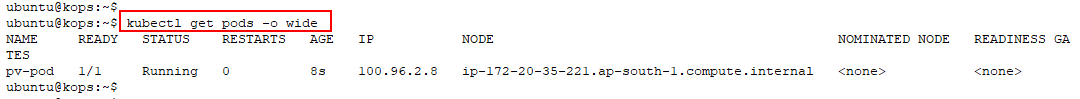
1. Apply the Pod yaml created in the previous step

$ kubectl apply -f pv-pod.yaml



1. View Pod details and see that is created on the required node

$ kubectl get pods -o wide



1. Access shell on a container running in your Pod

$ kubectl exec -it pv-pod -- /bin/bash

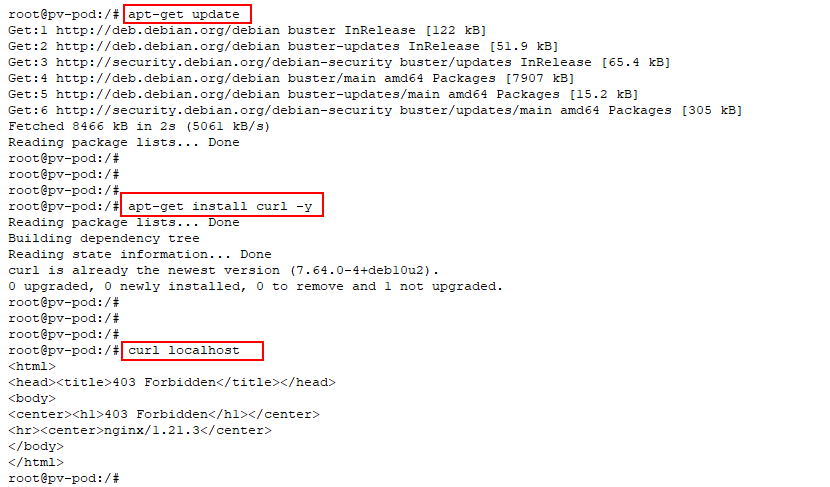


1. Run the following commands in the **container** to verify PersistentVolume

# apt-get update

# apt-get install curl -y

# curl localhost



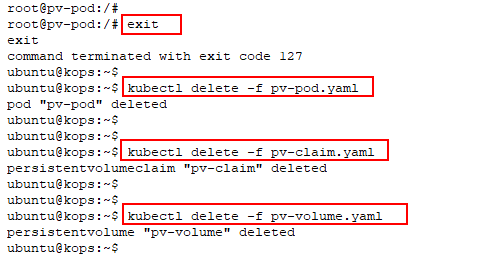
1. Exit the Pod and delete the resources created in this lab.

$ exit

$ kubectl delete -f pv-pod.yaml

$ kubectl delete -f pv-claim.yaml

$ kubectl delete -f pv-volume.yaml



# Chapter Four: Advanced Kubernetes

**Topics covered in this unit:**

* Controllers
* ReplicationController
* ReplicaSet
* Deployment
* DaemonSet

## Controllers - ReplicaSet

ReplicaSet is an advanced Replication Controller. The set-based selector is compatible with Replica Set, while Replication Controller only works with the equality-based selector.

Commands for both, ReplicaSet and Replication Controller are common most of the time.

ReplicaSets are most used along with Deployments to orchestrate and manage Pods. They can be run independently as well. When used with Deployment, no ReplicaSet management is necessary. It ensures a set number of Pod replicas are running at any given time. As Deployment is done at a much higher level, it is recommended to run ReplicaSets with Deployment. The block below shows an example of ReplicaSet:

apiVersion: apps/v1beta2

kind: ReplicaSet

metadata:

name: frontend

labels:

app: guestbook

tier: frontend

spec:

replicas: 3

selector:

matchLabels:

tier: frontend

matchExpressions:

- {key: tier, operator: In, values: [frontend]}

template:

metadata:

labels:

app: guestbook

tier: frontend

spec:

containers:

- name: php-redis

image: gcr.io/google\_samples/gb-frontend:v3

resources:

requests:

cpu: 100m

memory: 100Mi

env:

- name: GET\_HOSTS\_FROM

value: dns

ports:

- containerPort: 80

## Controller - Deployments

A Deployment controller delivers updates for Pods and Replica Sets.

The desired state can be described, and the Deployment controller makes sure the current state matches the described one.

Uses of Deployment:

* To roll out, ReplicaSet to create Pods and monitor its status
* Changing or updating Pod state
* Scaling up of the deployment
* Rollback to a previous Deployment
* Pause Deployment to fix the template
* Cleaning old ReplicaSets

The following is an example of deployment to bring up three nginx Pods:

apiVersion: apps/v1beta2

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

**Rollover**

When the Deployment controller notices a new Deployment object, it creates a ReplicaSet to bring up Pods as described. In case a Deployment is in progress, and an updated Deployment is applied, the replicas already launched are slowly scaled-down and killed, then the newer ones are created.

**Rollback**

A complete history is recorded, and rolling back is possible at any given time. The revision history of deployment is stored in the ReplicaSets controlled by it. Each old ReplicaSets is retained but the number of ReplicaSets to retain can be specified. Setting the number of ReplicaSets to retain to “0”, results in the removal of all old ReplicaSets.

Once an old ReplicaSet is deleted, it is not possible to roll back to that particular revision of deployment.

**Deployment Status**

A deployment goes through various stages of its lifecycle. It is said to be Progressing when rolling out a ReplicaSet, complete when the task is done, or when it fails.

A deployment is said to be Progressing when:

* Creating a new ReplicaSet
* Scaling up new ReplicaSets
* Scaling down old ReplicaSets
* New Pods are ready

A deployment is said to be complete when:

* All updates are done as described
* All new replicas are available
* All old replicas are killed

A deployment is said to have failed when:

* Insufficient permissions are set
* Errors in pulling image occur
* Quota is insufficient
* Application runtime misconfiguration

## DaemonSet

A DaemonSet runs a single copy of a Pod either in all Nodes or the Nodes that you specify. Pods are automatically added to newly added Nodes. When we remove a node from the cluster, these Pods are garbage collected. Deleting a DaemonSet object will delete all the Pods created by that DeamonSet.

Some use cases of DaemonSet are

* Configuring cluster storage daemon, such as glusterd, ceph etc. on each node
* Configuring log collection daemon like fluentd or logstash on each node.
* Configuring a node monitoring daemon on every node like Prometheus node exporter, collectd, New Relic agent.
* Cluster bootstrapping is another use case. The DaemonSet controller can create Pods even when the scheduler has not been started.

The following is an example of DaemonSet to bring up fluentd Pod on every Node:

apiVersion: extensions/v1beta1

kind: DaemonSet

metadata:

name: fluentd

namespace: default

labels:

app: fluentd

spec:

selector:

matchLabels:

app: fluentd

template:

metadata:

labels:

app: fluentd

spec:

containers:

- name: fluentd

image: fluent/fluentd

**Node Specific Daemons**

If we plan to have different logging, monitoring, or storage solutions on different nodes of the cluster or if we want to deploy the daemons only to a specific set of nodes instead of all Nodes, we can make use of the Node selector to specify a subset of nodes for the daemon set. DaemonSet controller will create Pods on nodes that match that node affinity.

**DaemonSet Update Strategy**

DaemonSet supports the Rolling update and OnDelete update.

* **OnDelete**: The default update strategy. The manual deletion of DaemonSet pods needs to be done after updating the DaemonSet template. (Kubernetes version 1.5 or before)
* **Rolling Update**: On updating a DaemonSet template, old DaemonSet pods are killed, and new DaemonSet pods are created automatically.

# Lab 4: Deployment

This lab demonstrates Deployments in Kubernetes.

Topics

* Write a Deployment yaml and apply it
* Update the Deployment with a newer image
* Rollback of Deployment
* Scaling of the Deployments

## Task 1: Write a Deployment yaml and Apply it

1. Create a **dep-nginx.yaml** using content given below

$ vi dep-nginx.yaml

Shape

Description automatically generated with low confidence

apiVersion: apps/v1

kind: Deployment

metadata:

name: nginx-dep

labels:

***app: nginx-dep***

spec:

replicas: 3

selector:

matchLabels:

***app: nginx-app***

template:

metadata:

labels:

***app: nginx-app***

spec:

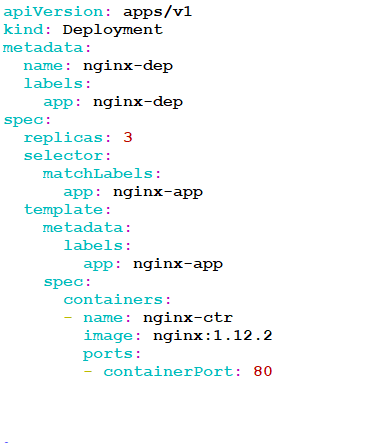
containers:

- name: nginx-ctr

image: nginx:1.12.2

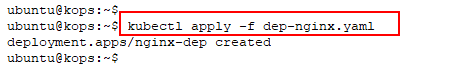
ports:

- containerPort: 80



1. Apply the Deployment yaml created in the previous step

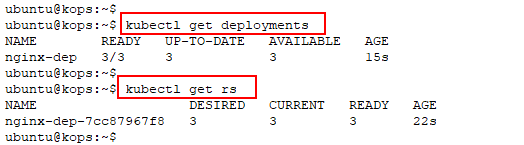
$ kubectl apply -f dep-nginx.yaml



1. View Deployment details and see that it is created as a Replica Set (rs)

$ kubectl get deployments

$ kubectl get rs



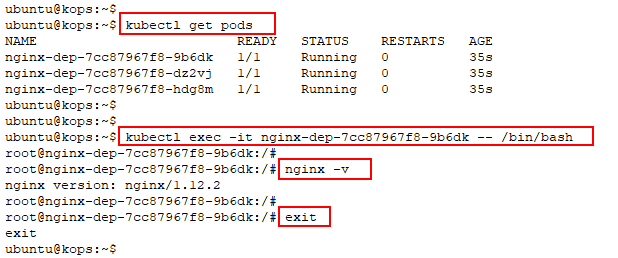
1. Access one of the Pods and view nginx version

$ kubectl get pods

$ kubectl exec -it **<pod\_name>** -- /bin/bash

# nginx -v

# exit



## Task 2: Update the Deployment with a Newer Image

1. Update the nginx image in Pods using Rolling updates

$ kubectl set image deployment/nginx-dep nginx-ctr=nginx:1.11

Graphical user interface

Description automatically generated with low confidence

1. Describe the deployment and see that the old pods are replaced with newer ones

$ kubectl describe deployments

Graphical user interface, text, application, email

Description automatically generated

A screenshot of a cell phone

Description automatically generated

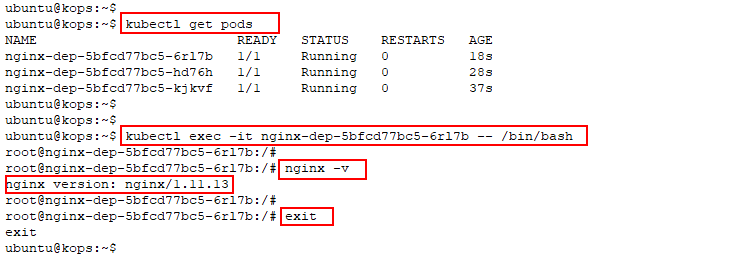
1. Access one of the Pods and view nginx version. As shown below, the version of nginx has changed

$ kubectl get pods

$ kubectl exec -it **<pod\_name>** -- /bin/bash

# nginx -v

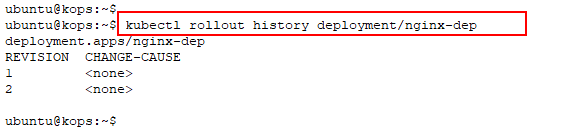
# exit



## Task 3: Rollback of Deployment

1. View the history of Deployments

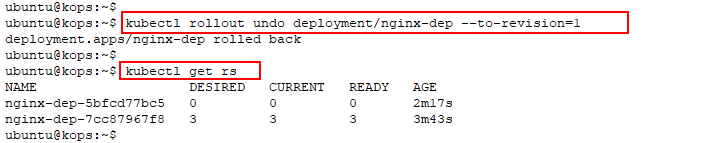
$ kubectl rollout history deployment/nginx-dep



1. Rollback the Deployment done in the previous task. Rollback to the first revision

$ kubectl rollout undo deployment/nginx-dep --to-revision=1

$ kubectl get rs



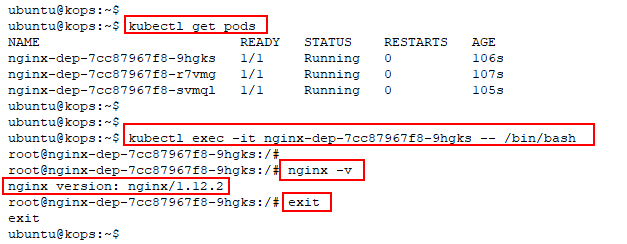
1. Access one of the Pods and view nginx version. As shown below, the version of nginx has changed back to what it was initially

$ kubectl get pods

$ kubectl exec -it **<pod\_name>** -- /bin/bash

# nginx -v

# exit

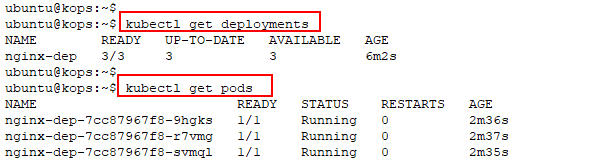


## Task 4: Scaling of Deployments

1. View the number of Pod replicas created by the Deployment

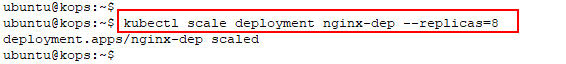
$ kubectl get deployments

$ kubectl get pods



1. Scale **up** the deployment to have 8 Pod replicas

$ kubectl scale deployment nginx-dep --replicas=8



1. Verify the scaling operation and see that number of Pod replicas are **8**

$ kubectl get deployments

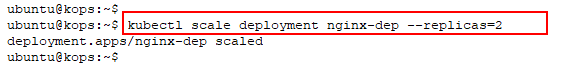
$ kubectl get pods

Table

Description automatically generated

1. Scale **down** the deployments to 2 Pod replicas

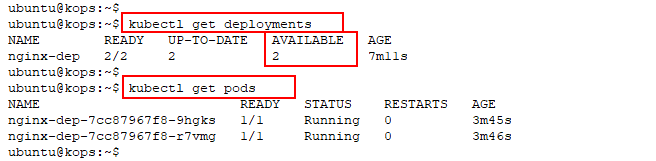
$ kubectl scale deployment nginx-dep --replicas=2



1. Verify the scaling operation and see that number of Pod replicas are **2**

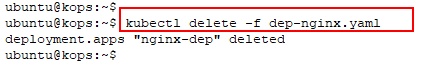
$ kubectl get deployments

$ kubectl get pods



1. Delete the resources created in this lab.

$ kubectl delete -f dep-nginx.yaml



# Lab 5: DaemonSet in Kubernetes

In this lab, we will demonstrate the functionality of daemonset.

1. Create a daemonset yaml called **ds-pod.yaml.** Enter the contents given below in the yaml

$ vi ds-pod.yaml



apiVersion: apps/v1

kind: DaemonSet

metadata:

name: fluent-ds

labels:

app: flentd-ds

spec:

selector:

matchLabels:

***app: fluentd-app***

template:

metadata:

labels:

***app: fluentd-app***

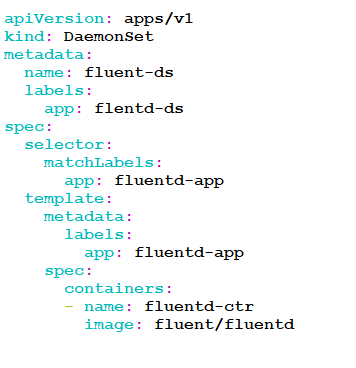
spec:

containers:

- name: fluentd-ctr

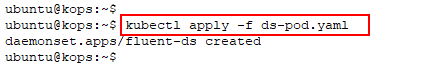
image: fluent/fluentd

Once entered, it should look as shown below



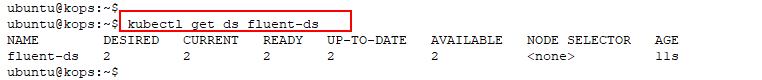
1. Create a daemonset object using the yaml created in the previous step

$ kubectl apply -f ds-pod.yaml



1. View the status of the daemonset

$ kubectl get ds fluent-ds



1. View the running pods and see that same pod is scheduled on both the nodes

$ kubectl get pods -o wide

Graphical user interface, application

Description automatically generated with medium confidence

1. Delete the daemonset.

$ kubectl delete -f ds-pod.yaml



# Lab 6: StatefulSet Implementation

## Task 1: Creating a StatefulSet

1. Download the yaml file that create the statefulset and view the configurations in the file. It basically creates a statefulset and service for it

$ wget files.cloudthat.training/devops/kubernetes-essentials/nginx-sts.yaml

Graphical user interface

Description automatically generated with medium confidence

1. Create the Headless Service and StatefulSet which from nginx-sts.yaml

$ kubectl apply -f nginx-sts.yaml

Text

Description automatically generated

1. Verify that service has been created. You can use either of the commands to view the service created

$ kubectl get service nginx-svc

Text

Description automatically generated

1. Verify that statefulset has been created

$ kubectl get statefulset nginx-sts

Text

Description automatically generated with medium confidence

1. Verify the ordered pod creation.

$ kubectl get pods -w -l app=nginx-sts

Text

Description automatically generated

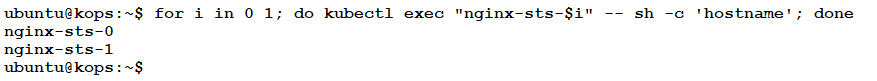
1. Fire the Above command again to verify the pods are up and running. The output will look like below

Text

Description automatically generated

1. Execute thehostname command in each Pod

$ for i in 0 1; do kubectl exec "nginx-sts-$i" -- sh -c 'hostname'; done



1. Launch busy box as dns-test container and get into it. We are going to test the connectivity from the busy box container. Use nslookup command to test the reachability to the pod nginx-sts-0 and nginx-sts-0 using sevice nginx-svc

$ kubectl run -i --tty --image busybox:1.28 dns-test --restart=Never –rm

# nslookup nginx-sts-0.nginx-svc

# nslookup nginx-sts-1.nginx-svc

# exit

Graphical user interface, text, application, email

Description automatically generated

1. Delete pods and wait for stateful to restart them. Open a separate tab and watch the process of ordered deletion and recreation

$ kubectl delete pod -l app=nginx-sts

$ kubectl get pod -w -l app=nginx-sts

Chart

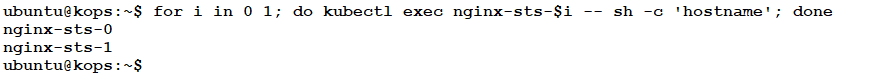
Description automatically generated

Text

Description automatically generated

1. View the pods hostname using the below command

$ for i in 0 1; do kubectl exec "nginx-sts-$i" -- sh -c 'hostname'; done



1. Verify whether the newly created pod are reachable using the below command. It bascicaly launch busybox as dns-test container and perform a nslookup for the newly created Pods

$ for i in 0 1; do kubectl exec "nginx-sts-$i" -- sh -c 'hostname'; done

# nslookup nginx-sts-0.nginx-svc

# nslookup nginx-sts-1.nginx-svc

# exit

Graphical user interface, text, application, email

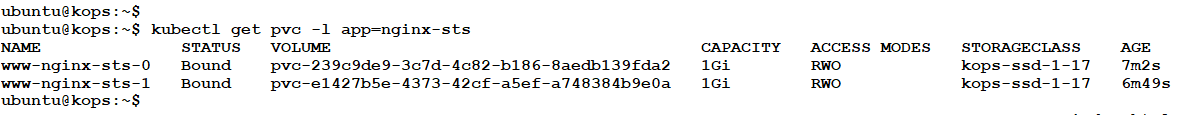
Description automatically generated

## Task 2: Writing to a Scalable Storage

The StatefulSet controller (i.e nginx-sts.yaml) created two PersistentVolumeClaims that are bound to two PersistentVolumes. The volume mount field in the yaml file is ensured that /usr/share/nginx/index.html is backed up by the persistent volume

1. Get the persistent volume chains for the pods **nginx-sts-0** and **nginx-sts-1**

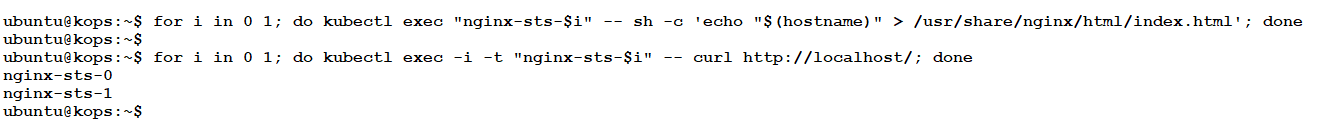
$ kubectl get pvc -l app=nginx-sts



1. Now, let us make the index.html file to contain the host name of the pods of the statefulset and curl the pods and see whether the changes are reflected

$ for i in 0 1; do kubectl exec "nginx-sts-$i" -- sh -c 'echo "$(hostname)" > /usr/share/nginx/html/index.html'; done

$ for i in 0 1; do kubectl exec -i -t "nginx-sts-$i" -- curl http://localhost/; done



## Task 3: Scaling StatefulSet

1. Scale the application to five replicas and watch creation process. Open a new tab and fire below commands and watch the process of scaling the replicas.

$ kubectl scale sts nginx-sts --replicas=5

$ kubectl get pods -w -l app=nginx-sts

Text

Description automatically generated

Table

Description automatically generated

1. Verify that the scaled replicas are up and running

$ kubectl get pods -w -l app=nginx-sts

Text

Description automatically generated

1. Get the pvc. Note that the PVC has also increased.

$ kubectl get pvc -l app=nginx-sts

A screenshot of a computer

Description automatically generated with medium confidence

1. Scale down statefulset and verify ordered termination. Perform a edit on the statefullset. Look for the replicas and replace the value from 5 to 3. Save and close the editor. This command automatically applies the changes to the statefulset file

$ kubectl edit sts nginx-sts

Graphical user interface, text, application, email

Description automatically generated

Text

Description automatically generated with medium confidence

$ kubectl get pods -w -l app=nginx-sts

Notice that the controller deletes the pods one at a time. It waits for one to completely shut down before going to next

Table

Description automatically generated

1. Verify statefulSet’s PersistentVolumeClaims and verify that are not deleted on scaling down.

$ kubectl get pvc -l app=nginx-sts

A screenshot of a computer

Description automatically generated with medium confidence

## Task 4 : Clean-up Resources

1. Delete the statefulset and Service

$ kubectl delete -f nginx-sts.yaml

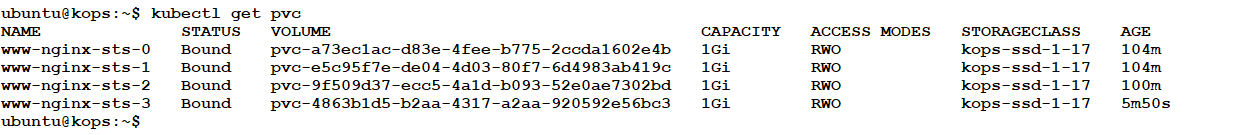
Text

Description automatically generated

1. List all the PV and PVC’s that has been allocated to Statefulset pods and delete them:

$ kubectl get pvc

$ kubectl delete pvc --all



Text

Description automatically generated

# Chapter Five: Advanced Kubernetes Administration

**Topics covered in this unit:**

* Assigning Pods to Nodes
* Node Affinity and Anti-affinity
* Pod Affinity and Anti-Affinity
* Node Maintenance
* Resource Quota
* Troubleshooting

## Assigning Pods to Nodes

In Kubernetes, it is possible to restrict a Pod to run on a specific Node. This is achieved by using Label-Selectors. Such restrictions are often unnecessary as the scheduler does an optimal placement of Pods. It is used in rear circumstances when specific Pods are to be run on certain Nodes.

**NodeSelector**

NodeSelector is the simplest constraint. It is a part of PodSpec and specifies a key-value pair. For a Pod to be allowed to run a Node, it must have each key-value pair as Label.

NodeSelector provides a simple way to constrain Pods to Nodes using Labels. It is made more powerful by using the affinity/anti-affinity feature which provides a better set of options for applying constraints.

* The language used is more meaningful and does not have to be an exact match
* The rule can be set as a soft requirement or as a preference instead of a hard requirement
* Constraints can be made based on Labels on other Pods on the Node rather than on the same Pod

Below yaml shows an example of nodeSelector being used to assign a pod to a node

apiVersion: v1

kind: Pod

metadata:

name: nginx

labels:

env: test

spec:

containers:

- name: nginx

image: nginx

imagePullPolicy: IfNotPresent

nodeSelector:

disktype: ssd

**NodeAffinity**

Node affinity is similar to nodeSelector in terms of functionality, but it also has the first two features mentioned above.

It is of two types:

* **RequiredDuringSchedulingIgnoredDuringExecution**: It can be considered as a soft constraint.
* **PreferredDuringSchedulingIgnoredDuringExecution**: It can be considered as a hard constraint.

**PodAffinity**

Pod affinity allows the user to constrain pods to nodes based on the labels of the pods already present on the nodes.

Pod affinity is of two types: Inter-pod and anti-affinity.

Below is an example of **podAntiAffinity**.

apiVersion: apps/v1

kind: Deployment

metadata:

name: nginx

spec:

selector:

matchLabels:

app: store

replicas: 3

template:

metadata:

labels:

app: nginx

spec:

affinity:

podAntiAffinity:

requiredDuringSchedulingIgnoredDuringExecution:

- labelSelector:

matchExpressions:

- key: app

operator: In

values:

- nginx

topologyKey: "kubernetes.io/hostname"

containers:

- name: nginx

image: nginx

## Node Maintenance

In case we need to perform kernel upgrade, libc upgrade or hardware repair, etc., with low downtime, then we need to restart the node. The pods that are scheduled with the node is also restarted when kubelet restarts. In case the reboot takes longer than usual (which is 5 minutes by default) then pods are terminated that are bond to the unavailable nodes. A new copy of the pods will be started on a differently available node using their replica sets, as we have replica sets for all the available pods, so that the update can be done without any special coordination.

To have more control over the upgrading process, the following workflow can be used:

You can use “kubectl drain” command which terminates all pods on the node and makes the respective node unschedulable.

**kubectl drain $NODENAME**

The nodes which are terminated by “kubectl drain” are marked un-schedulable, and it prevents the new pods from arriving at the nodes getting drained. It also deletes all the pods except the mirror pods, as it is not possible to delete mirror pods through the API server.

**Kubectl cordon $NODENAME**

This command makes the node as un-schedulable, and it prevents new pods from being scheduled to that node without affecting the existing pods that are already running on the node.

**kubectl uncordon $NODENAME**

When the node’s VM instance is deleted, and you created a new VM instance, then a new schedulable node resource is created automatically. (if you’re with a cloud provider that supports node discovery; presently, this is only Google Compute Engine, not including CoreOS on Google Compute Engine using kube-register.

If the Kubernetes node requires security or any other kind of patches, we can do so by firstly evicting all the Pods that have been running on the nodes gracefully, i.e., PodDistributionBudget that you have specified will be taken into consideration. The command for evicting the Pods safely out of the Nodes is “kubectl drain <node-name>. Once the patches have been applied, the evicted Pods can be rescheduled onto the same node.

## Resource Quotas and Limits

When a cluster is shared among several teams, it becomes necessary to limit the amount of resources that can be used by them. This is achieved by the object ResourceQuotas in Kubernetes. It can limit the amount of resource consumption per namespace. It can be used to limit the quantity of objects and the amount of compute resources per namespace.

Resource quotas are independent of cluster capacity.

The quota system keeps track of resources created by the users and ensures that the hard limits are not exceeded.

Compute resources like CPU and Memory can be limited in a namespace.

The below example shows a **ResourceQuota** for namespace **quotas.** It specifies hard limits of 2 CPUs and 2Gi memory and requests of 1 CPU and 1Gi memory.

apiVersion: v1

**kind: ResourceQuota**

metadata:

name: quotaname

**namespace: quotans**

spec:

hard:

requests.cpu: "1"

requests.memory: 1Gi

limits.cpu: "2"

limits.memory: 2Gi

**Requests and Limits**

When compute resources are to be allocated to a container, it may specify request and limit values. The request is the minimum amount of resources that are available for a container, and limits are the maximum amount of resources that are to be used.

The below example shows a Pod yaml with limits of 1000m CPU and 800Mi memory and requests of 350m CPU and 600Mi memory.

apiVersion: v1

**kind: Pod**

metadata:

name: quota-pod

**namespace: quotans**

spec:

containers:

- name: quota-container

image: nginx

resources:

limits:

memory: "800Mi"

cpu: "1000m"

requests:

memory: "600Mi"

cpu: "350m"

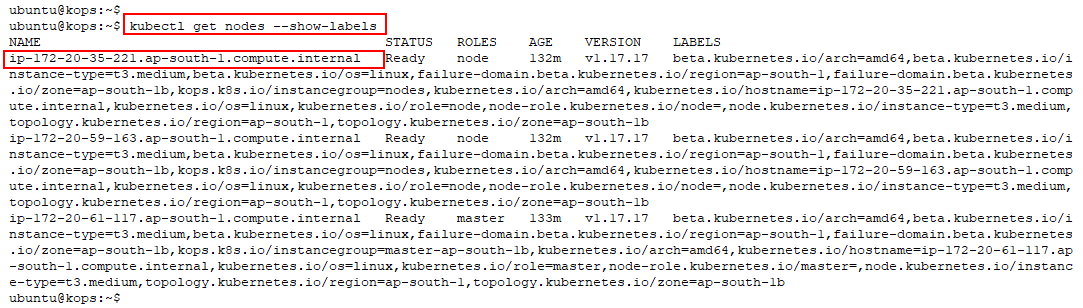
# Lab 7: Node Labelling and Constraining pods in Kubernetes

This lab demonstrates the labeling of nodes and constraining pod scheduling using nodeSelector.

## Task 1: Node labeling and constraining pods

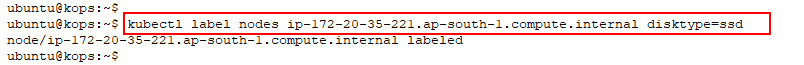
1. View the default labels for all the nodes. Make a note of the **NAME** of one of the nodes. This node will be used to schedule a pod

$ kubectl get nodes --show-labels



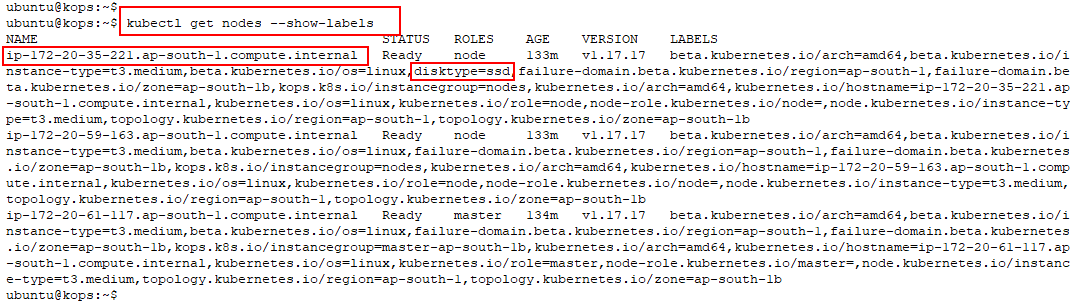
1. Add a label to the node

$ kubectl label nodes **<node\_name>** disktype=ssd



1. View the labels again to verify labeling from the previous step

$ kubectl get nodes --show-labels



1. Create a pod and use nodeSelector to constrain it to a particular node

$ vi nlns-pod.yaml

Shape

Description automatically generated with low confidence

Enter the contents of the yaml as shown below

apiVersion: v1

kind: Pod.

metadata:

name: nlns-nginx-pod

labels:

env: test

spec:

containers:

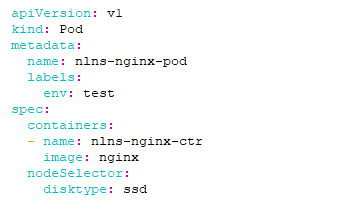
- name: nlns-nginx-ctr

image: nginx

nodeSelector:

disktype: ssd

Once entered, the file should look as shown below



1. Create the pod using the yaml created in the previous step

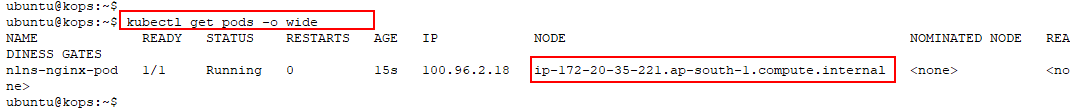
$ kubectl apply -f nlns-pod.yaml

A picture containing logo

Description automatically generated

1. Verify that the pod is scheduled on the desired node

$ kubectl get pods -o wide

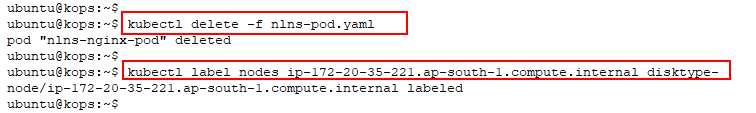


## Task 2: Cleanup

1. Delete the objects created in this lab

$ kubectl delete -f nlns-pod.yaml

$ kubectl label nodes **<node\_name>** disktype-

****

1. Verify that the label is deleted.

$ kubectl get nodes --show-labels | grep ssd



# Lab 8: Advanced Pod Scheduling

This lab demonstrates advanced Pod scheduling features in Kubernetes.

## Task 1: Node Affinity

1. Create a nginx yaml of **preferredDuringScheduling** nodeAffinity using content given below

$ vi aff-na-pod1.yaml

A picture containing shape

Description automatically generated

apiVersion: v1

kind: Pod

metadata:

name: na-nginx-pod1

spec:

containers:

- name: na-nginx-ctr1

image: nginx

affinity:

nodeAffinity:

preferredDuringSchedulingIgnoredDuringExecution:

- weight: 1

preference:

matchExpressions:

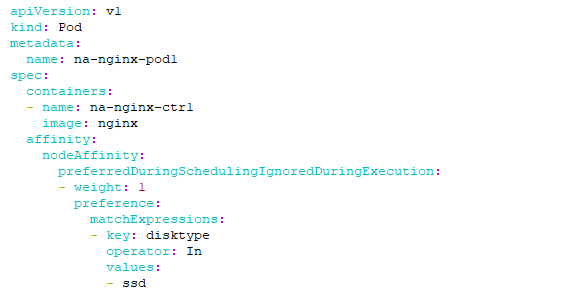
- key: disktype

operator: In

values:

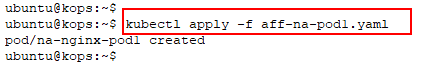
- ssd

Once entered, it should look as shown below



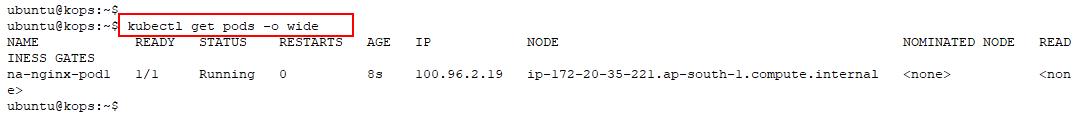
1. Create a pod using the yaml created in the previous step

$ kubectl apply -f aff-na-pod1.yaml



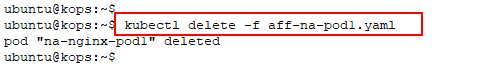
1. View pods and notice that it has been created despite none of the nodes having the specified label

$ kubectl get pods -o wide



1. Delete the pod **na-nginx-pod1**

$ kubectl delete -f aff-na-pod1.yaml



1. Create another pod with nodeAffinity type of **requiredDuringScheduling** using content given below

$ vi aff-na-pod2.yaml



apiVersion: v1

kind: Pod

metadata:

name: na-nginx-pod2

spec:

containers:

- name: na-nginx-ctr2

image: nginx

affinity:

nodeAffinity:

***requiredDuringSchedulingIgnoredDuringExecution:***

nodeSelectorTerms:

- matchExpressions:

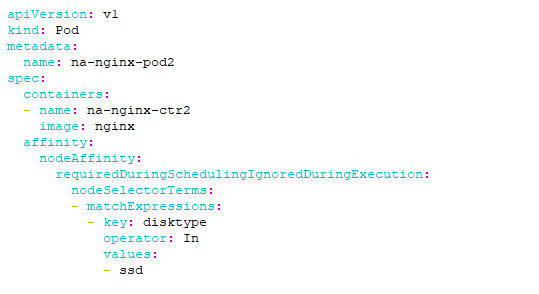
- key: disktype

operator: In

values:

- ssd

Once entered, it should look as shown below



1. Create a pod using the yaml created in the previous step

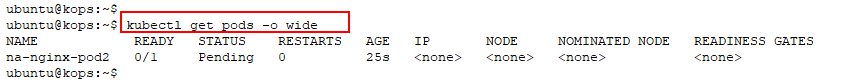
$ kubectl create -f aff-na-pod2.yaml

A picture containing graphical user interface

Description automatically generated

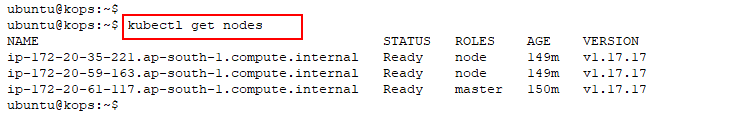
1. View pods and see that it is **not** scheduled as any of the nodes have the specified label

$ kubectl get pods -o wide



1. View all the nodes in the cluster and make a note of the name of one of the **nodes**

$ kubectl get nodes



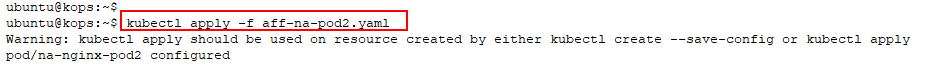
1. Using the node name from the previous step, add a label to one of the nodes

$ kubectl label nodes **<node\_name>** disktype=ssd



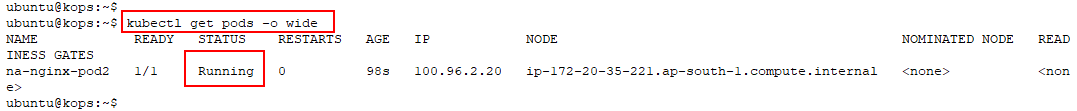
1. Apply the yaml again as it is stuck in a **pending** state

$ kubectl apply -f aff-na-pod2.yaml



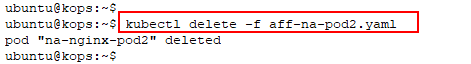
1. View the pods again and see that it is now scheduled

$ kubectl get pods -o wide



1. Delete the pod

$ kubectl delete -f aff-na-pod2.yaml



## Task 2: Pod Affinity

1. Create a yaml file with the content given below. Note that the label used is **app: nginx**

$ vi aff-pa-pod1.yaml



apiVersion: v1

kind: Pod

metadata:

name: pa-nginx-pod1

labels:

app: pa-nginx-app

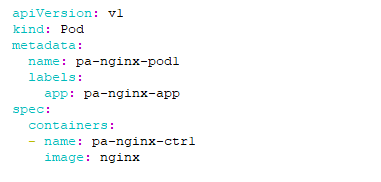
spec:

containers:

- name: pa-nginx-ctr1

image: nginx

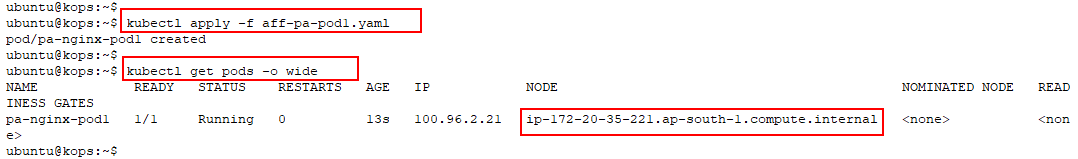
Once entered, the file should look as shown below



1. Create a pod using the yaml created in the previous step. View the node on which it was scheduled

$ kubectl apply -f aff-pa-pod1.yaml

$ kubectl get pods -o wide



1. Create a new pod yaml using the content given below

$ vi aff-pa-pod2.yaml



apiVersion: v1

kind: Pod

metadata:

name: pa-nginx-pod2

spec:

containers:

- name: pa-nginx-ctr2

image: nginx

affinity:

podAffinity:

requiredDuringSchedulingIgnoredDuringExecution:

- labelSelector:

matchExpressions:

- key: app

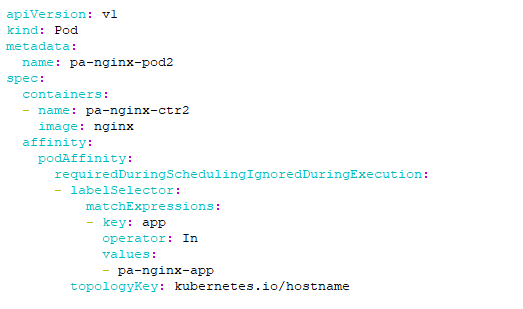
operator: In

values:

- pa-nginx-app

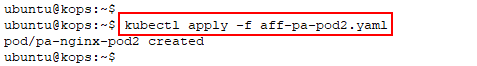
topologyKey: kubernetes.io/hostname

Once entered, the file should look as shown below



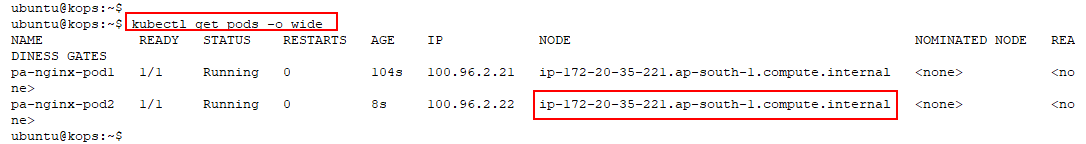
1. Apply the yaml created in the previous step

$ kubectl apply -f aff-pa-pod2.yaml



1. View on which node the pod is scheduled. Notice that the pod is scheduled on the same node as the pod created previously

$ kubectl get pods -o wide

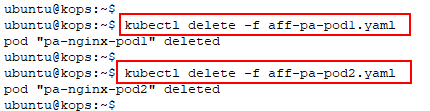


## Task 3: Clean-up

1. Delete the pods created in the previous task.

$ kubectl delete -f aff-pa-pod1.yaml

$ kubectl delete -f aff-pa-pod2.yaml



# Lab 9: Resource Quotas in Kubernetes

This lab demonstrates how **resourcequotas** can be used to restrict resources within a namespace.

Topics

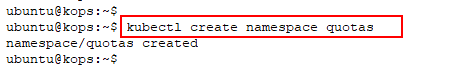
* Creating a namespace
* Creating a resourcequota
* Verify resourcequota functionality

## Task 1: Creating a Namespace

In this task, a namespace called **quotas** is created.

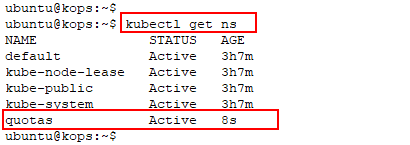
1. Create a namespace

$ kubectl create namespace quotas



1. Verify namespace creation

$ kubectl get ns



## Task 2: Creating a resourcequota

In this task, a resourcequota yaml file is created, and it is applied to the namespace created in the previous step.

1. Create a file **quota.yaml**

$ vi rq-quotas.yaml



Enter the content given below

apiVersion: v1

kind: ResourceQuota

metadata:

name: quota

namespace: quotas

spec:

hard:

requests.cpu: "1"

requests.memory: 1Gi

limits.cpu: "2"

limits.memory: 2Gi

Once entered, it should look as shown

Text

Description automatically generated

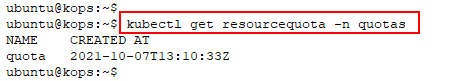
1. Create the **resourcequota** from the yaml

$ kubectl create -f rq-quotas.yaml



1. Verify resourcequota creation

$ kubectl get resourcequota -n quotas



## Task 3: Verify resourcequota Functionality

In this task, two pods exceeding the request quota are created to demonstrate the functionality of resourcequotas.

1. Create a pod yaml called **quota-pod.yaml**

$ vi rq-pod.yaml



Enter the below content in the yaml file

apiVersion: v1

kind: Pod

metadata:

name: quota-pod

namespace: quotas

spec:

containers:

- name: quota-ctr

image: nginx

resources:

limits:

memory: "800Mi"

cpu: "1000m"

requests:

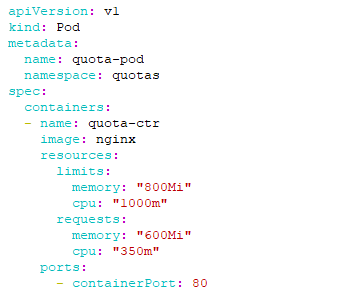
memory: "600Mi"

cpu: "350m"

ports:

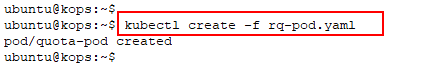
- containerPort: 80

Once entered, it should look as shown below



1. Create the pod using the yaml created in the previous step

$ kubectl create -f rq-pod.yaml



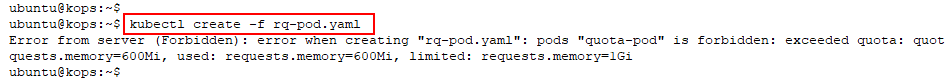
1. Once the pod is created, view the resources used by the pod

$ kubectl get resourcequota -n quotas -o yaml



1. Create the same pod again and see that it does not get created due to the set resourcequotas. Two pods together will request **1.2Gi** of memory while quota is set at **1Gi**

$ kubectl create -f rq-pod.yaml



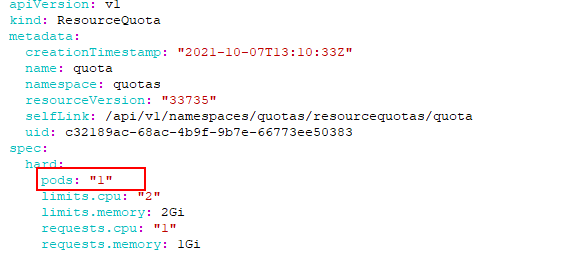
## Task 4: Limiting Number of Pods

In this task, the functionality of resourcequotas for a number of pods is demonstrated.

1. Edit the existing resourcequota and include a limit for the number of pods

$ kubectl edit resourcequotas quota -n quotas

1. Append **pods: “1”** to the limits in the file as shown below



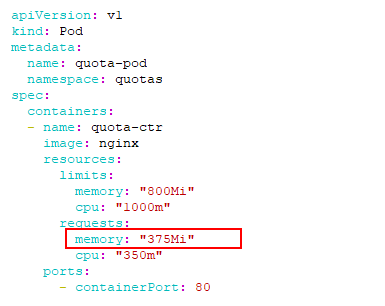
If the editing is successful, the below output can be shown



1. Modify the pod memory limit to ensure that pod creation is not affected by the memory limit

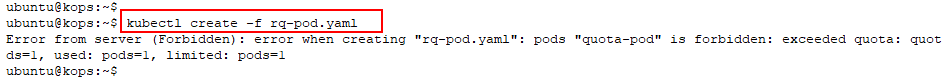
$ vi rq-pod.yaml





1. Now, try to create a pod and note that it will not be created due to the restriction on the number of pods

$ kubectl create -f rq-pod.yaml



## Task 5: Clean-up

1. Delete the quota to clean up.

$ kubectl delete ns quotas



# Lab 10: Node Maintenance

## Task 1: Node Maintenance

This lab demonstrates disabling and enabling scheduling of Pod deployments on a Node and Evicting Pods from a Node.

1. Use the **dep-nginx.yaml** file created in Lab 4 and verify the contents are same as below.

$ vi dep-nginx.yaml



apiVersion: apps/v1

kind: Deployment

metadata:

name: nginx-dep

labels:

app: nginx-dep

spec:

replicas: 3

selector:

matchLabels:

app: nginx-app

template:

metadata:

labels:

app: nginx-app

spec:

containers:

- name: nginx-ctr

image: nginx:1.12.2

ports:

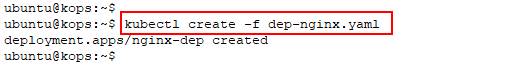
- containerPort: 80

Once completed, the file should look as shown below



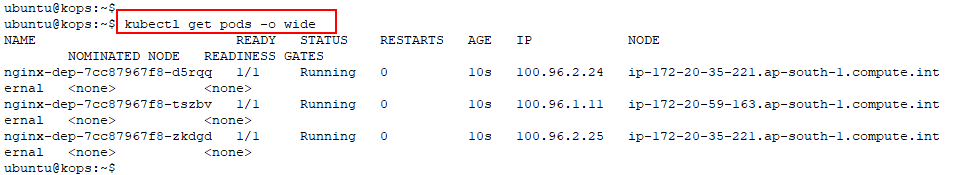
1. Create a deployment using the following command

$ kubectl create -f dep-nginx.yaml



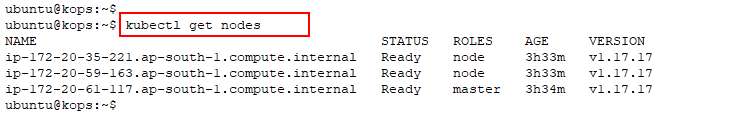
1. Type the following command to get the details of the Pods, and node on which they are running

$ kubectl get pods -o wide



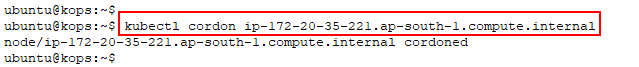
1. Get the details of the node with the following command

$ kubectl get nodes



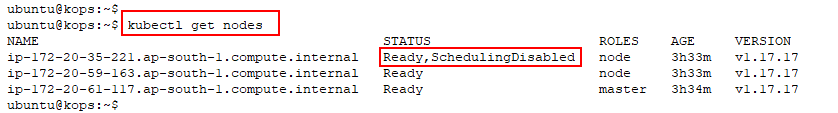
1. Cordon a node with the following command

$ kubectl cordon <node-name>



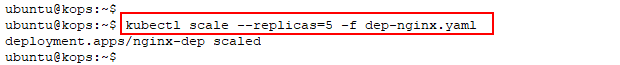
1. Check the status of the nodes to see if it is uncordoned

$ kubectl get nodes



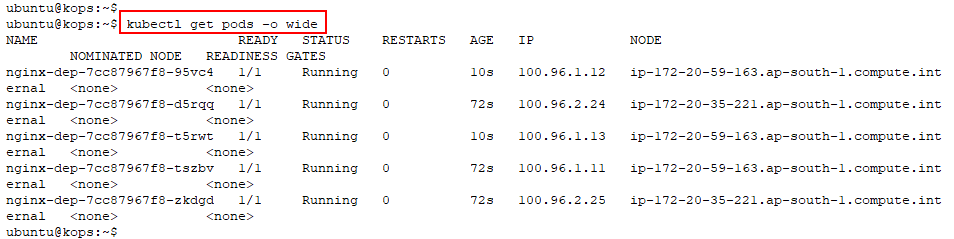
1. Scale your deployment now to 5 replicas. The new pod will not get deployed on the cordoned node

$ kubectl scale --replicas=5 -f dep-nginx.yaml



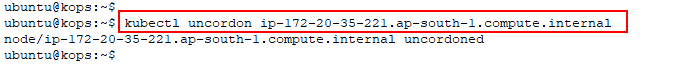
1. Use the following command to check the nodes on which the new Pod got deployed

$ kubectl get pods -o wide



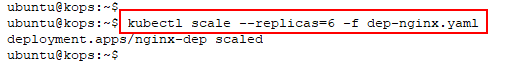
1. Use the following command to reschedule new containers on the same pod

$ kubectl uncordon <node-name>



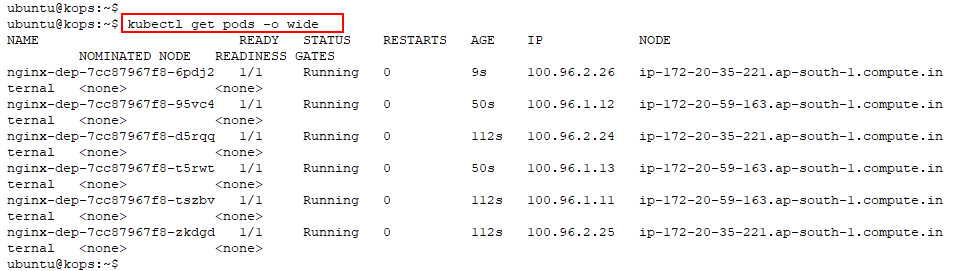
1. Scale your deployment with the following command

$ kubectl scale --replicas=6 -f dep-nginx.yaml



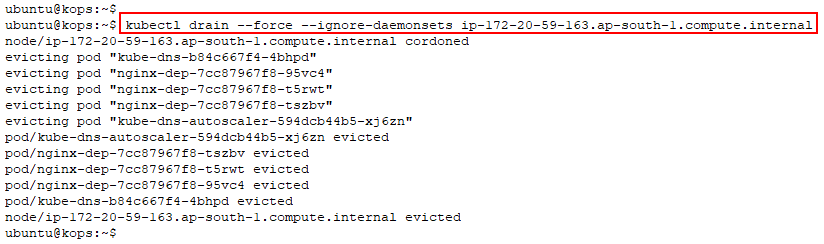
1. To check whether new pods get scheduled on the same pods, use the following command

$ kubectl get pods -o wide



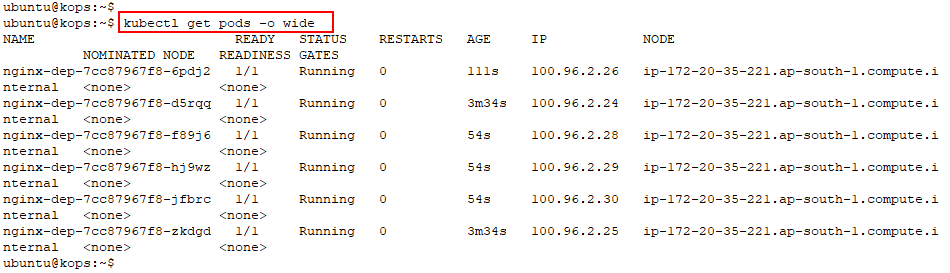
1. Drain a node with the following command. It will evict the pods from the specified node

$ kubectl drain --force --ignore-daemonsets <node-name>



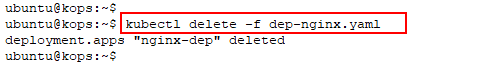
1. Use the following command to see if the pods have been successfully evicted from the node

$ kubectl get pods -o wide



1. Delete the deployment with the following command.

$ kubectl delete -f dep-nginx.yaml



## Task 2: Delete the cluster

1. Delete the cluster by using the below command

$ kops delete cluster **<cluster-name>** --yes

Graphical user interface, text, application

Description automatically generated

Text

Description automatically generated

1. Now go to the AWS console and delete the S3 bucket that was created for kops state storage, and terminate the kops machine

# Release Notes

|  |  |
| --- | --- |
| 16th April 2018 | * Moved labs from Heptio to Kops * Revised labs and added 7 labs * Added new chapter: Advanced Kubernetes Administration |
| 27th August 2018 | * Updated the labs. |
| 10th October 2018 | * Updated the labs. |
| 12th November 2018 | * Evaluated the entire course. |
| 11th July 2019 | * Updated the versions |
| 4th October 2019 | * Reviewed the labs |
| 29th May 2020 | * Updated the labs |
| 21st July 2020 | * Revamped the entire course |
| 7th Oct 2021 | * Revamped labs |