Logo

Description automatically generated

Kubernetes Application Development

Copyright @2012-23, CloudThat Technologies Private Limited, All Rights Reserved. Do not copy or distribute.



Table of Contents

[**Kubernetes Application Development** 4](#_Toc130824838)

[**About CloudThat** 4](#_Toc130824839)

[**About the Author** 4](#_Toc130824840)

[**Copyright** 5](#_Toc130824841)

[Lab 1: Bootstrap a Kubernetes Cluster using Kubeadm 2](#_Toc130824842)

[Topics 2](#_Toc130824843)

[Task 1: Launching Instances on AWS 2](#_Toc130824844)

[Task 2: Setting up Machines 9](#_Toc130824845)

[All steps in this task are to be performed on both the machines. 9](#_Toc130824846)

[Task 3: Initializing the Cluster 12](#_Toc130824847)

[Task 4: Joining a Cluster 14](#_Toc130824848)

[Task 5: Deploy Container Networking Interface 16](#_Toc130824849)

[Task 6: Create Pods 17](#_Toc130824850)

[Introduction to Jobs and CronJobs in Kubernetes 20](#_Toc130824851)

[Lab 2: Jobs and Cronjobs in Kubernetes 22](#_Toc130824852)

[Task 1: Jobs 22](#_Toc130824853)

[Task 2: Cronjobs 25](#_Toc130824854)

[Introduction to Sidecar Container 29](#_Toc130824855)

[Lab 3: Sidecar container 31](#_Toc130824856)

[Task:1 Creating an Application container with Sidecar container 31](#_Toc130824857)

[Introduction to Canary Deployment 34](#_Toc130824858)

[Lab 4 Canary Deployment on Kubernetes 35](#_Toc130824859)

[Liveness, Readiness and Startup Probes 41](#_Toc130824860)

[Lab 5: Liveness and Readiness probes 45](#_Toc130824861)

[Task 1: Liveness probes 45](#_Toc130824862)

[Task 2: Readiness probe 47](#_Toc130824863)

[Concept of Monitoring and Logging 53](#_Toc130824864)

[Logging 53](#_Toc130824865)

[Resource Quota 54](#_Toc130824866)

[Lab 6: Resource Quotas in Kubernetes 56](#_Toc130824867)

[Task 1: Creating a Namespace 56](#_Toc130824868)

[Task 2: Creating a resourcequota 57](#_Toc130824869)

[Task 3: Verify resourcequota Functionality 59](#_Toc130824870)

[Task 4: Limiting Number of Pods 61](#_Toc130824871)

[Task 5: Clean-up 62](#_Toc130824872)

[ConfigMaps and Secrets 63](#_Toc130824873)

[Lab 7: ConfigMap 64](#_Toc130824874)

[Task 1: Setting container environment variables using configmap 64](#_Toc130824875)

[Task 2: Setting configuration file with volume using ConfigMap 68](#_Toc130824876)

[Task 3: Creating a Secret 70](#_Toc130824877)

[Security Context 71](#_Toc130824878)

[Lab 8: Pod and Container level security using Context. 72](#_Toc130824879)

[Task 1: Set the security context for a pod 72](#_Toc130824880)

[Task 2: Set the security context for a container 75](#_Toc130824881)

[Network Policy 77](#_Toc130824882)

[Lab 9: Kubernetes Network Policy 2](#_Toc130824883)

[Task 1: Network policy with Pod labels 2](#_Toc130824884)

[Concept of Pod Scheduling 6](#_Toc130824885)

[Lab 10: Node Labelling and Constraining pods in Kubernetes 18](#_Toc130824886)

[Task 1: Node labeling and constraining pods 18](#_Toc130824887)

[Task 2: Cleanup 21](#_Toc130824888)

[Lab 11: Advanced Pod Scheduling 22](#_Toc130824889)

[Task 1: Node Affinity 22](#_Toc130824890)

[Task 2: Pod Affinity 28](#_Toc130824891)

[Task 3: Clean-up 32](#_Toc130824892)

[Introduction to Helm 33](#_Toc130824893)

[Lab 12: Installing WordPress with Helm 34](#_Toc130824894)

[Task 1: Helm setup 34](#_Toc130824895)

[Task 2: Setup WordPress 35](#_Toc130824896)

[Task 3: Verify that WordPress has been set up 38](#_Toc130824897)

[Task 4: Cleanup 41](#_Toc130824898)

# **Kubernetes Application Development**

## **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 @2012-23, CloudThat. All Rights Reserved.**

The contents of this course, its modules, and its related materials, including handouts to participants, are Copyright @2012-23, 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.

# Lab 1: Bootstrap a Kubernetes Cluster using Kubeadm.

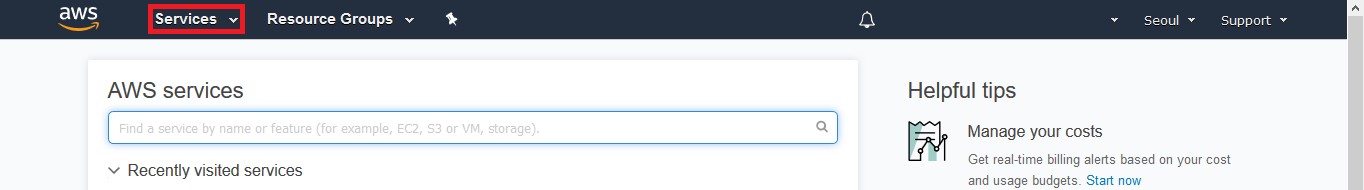
Two EC2 instances are used to set up Kubernetes master and node and setup using kubeadm.

### Topics

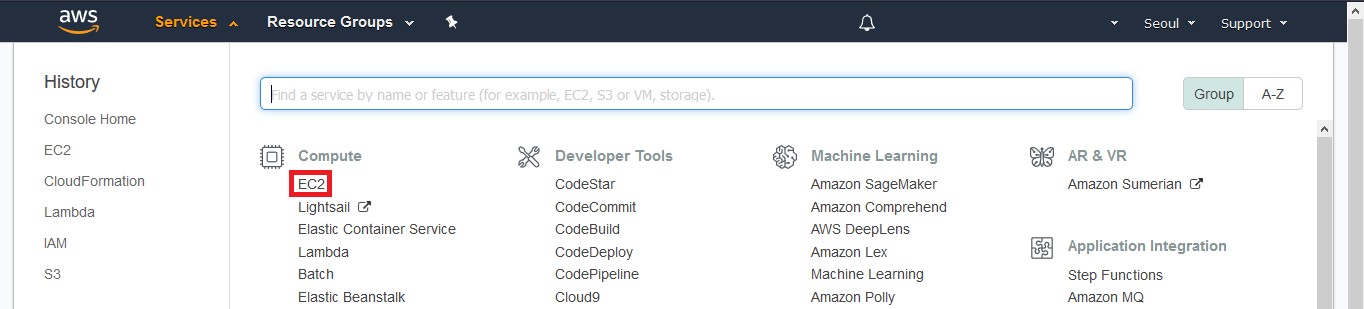
* Launching instances on AWS
* Setup master and node
* Initialize master
* Joining a cluster
* Deploy container network interface
* Deploy pod

## Task 1: Launching Instances on AWS

1. Click **Services** drop-down on the AWS console.



1. Then click **EC2.**



1. Create security group to open ports for internal communication.

Graphical user interface, application, Word

Description automatically generated

1. On the next page, click **Create Security Group.**

Graphical user interface, application

Description automatically generated

1. On the next page, enter **basic details i.e name and save it.**

Graphical user interface, application

Description automatically generated

1. On the next page, click **Edit Inbound Rules.**

Graphical user interface, text, application

Description automatically generated

1. **Graphical user interface, application

   Description automatically generated**On the next page, click **add inbound rules for same security group that we created.**
2. On the next page, click **Launch Instance.**

Graphical user interface, application

Description automatically generated

1. Configure virtual machine and create 3 VMS at single go.

Graphical user interface, application

Description automatically generated

1. Select **Instance type t2.medium and key**.

Graphical user interface, text, application

Description automatically generated

1. Select **Security Group that we created with name kubeadm-sc and click launch instance**.

Graphical user interface, text, application, email

Description automatically generated

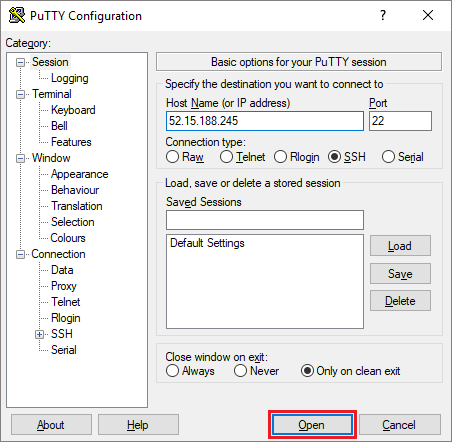
1. Skip the **Tags** page. Following ports are required to enable internally in master and worker nodes. Instead of opening all ports you can open these ports internally.

|  |  |  |
| --- | --- | --- |
| **Nodes** | **Port Number** | **Use Case** |
| Master, Workers | 2379 | etcd Client API |
| Master, Workers | 2380 | Etcd Server API |
| Master | 6443 | Kubernetes API Server (Secure Port) |
| Master, Workers | 6782 – 6784 | Weave Net Server/Client API |
| Master, Workers | 10250 – 10255 | Kubelet Communication |
| Workers | 30000 – 32767 | Reserved of NodePort Ips |

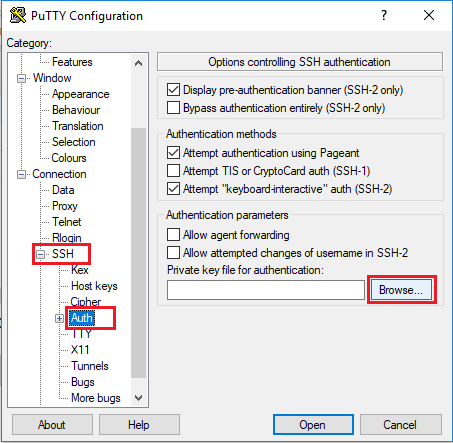
## Task 2: Setting up Machines

### All steps in this task are to be performed on both the machines.

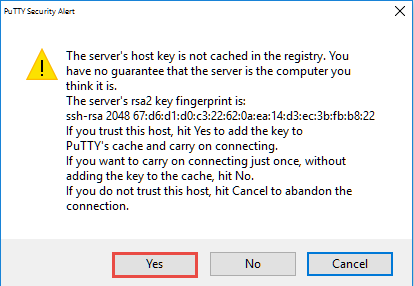
1. In **PuTTY Configuration** window, Paste your EC2 **Public IP or Public DNS** of control node in the **Hostname (or IP Address)** field.



1. Then expand the **SSH** section and select **Auth**. Click on the **Browse** button to select your **kubeadm.ppk** file and click **Open.**



1. Click **Yes** on the PuTTY Security Alert window.



1. Enter the username as **ubuntu.**



1. Switch to root.

$ sudo su



1. Create **kubeadm.sh** script file to install and configure kubeadm on **all the 3**

instances.

vi kubeadm-script.sh

#!/bin/bash

sudo apt-get update

sudo apt-get install apt-transport-https ca-certificates curl gnupg-agent software-properties-common -y

sudo apt install docker.io -y

sudo usermod -aG docker ubuntu

sudo cat > /etc/docker/daemon.json <<EOF

{

"exec-opts": ["native.cgroupdriver=systemd"],

"log-driver": "json-file",

"log-opts": {

"max-size": "1024m"

},

"storage-driver": "overlay2"

}

EOF

sudo systemctl daemon-reload

sudo systemctl restart docker

sudo echo "Environment=cgroup-driver=systemd/cgroup-driver=cgroupfs" >> /etc/systemd/system/kubelet.service.d/10-kubeadm.conf

sudo curl -s https://packages.cloud.google.com/apt/doc/apt-key.gpg | apt-key add -

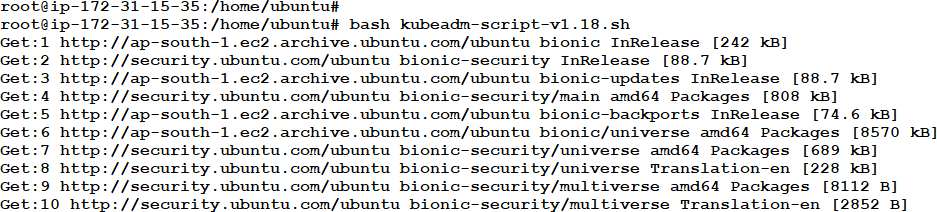
sudo echo 'deb http://apt.kubernetes.io/ kubernetes-xenial main' >> /etc/apt/sources.list.d/kubernetes.list

sudo apt-get update

sudo apt-get install -y kubelet=1.23.0-00 kubeadm=1.23.0-00 kubectl=1.23.0-00

1. Run the script to setup and configure kubeadm on **all 3** instances.

# bash kubeadm-script.sh



## Task 3: Initializing the Cluster

1. Set the hostname to all the three nodes as master, worker1, worker2 in their respective terminals for easy understanding, by running the below command:

# hostnamectl set-hostname master



# hostnamectl set-hostname worker1



# hostnamectl set-hostname worker2

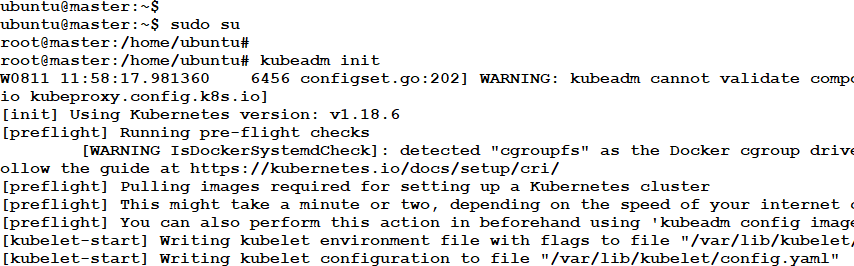


Note: Please take a new putty session to reflect the changed hostname.

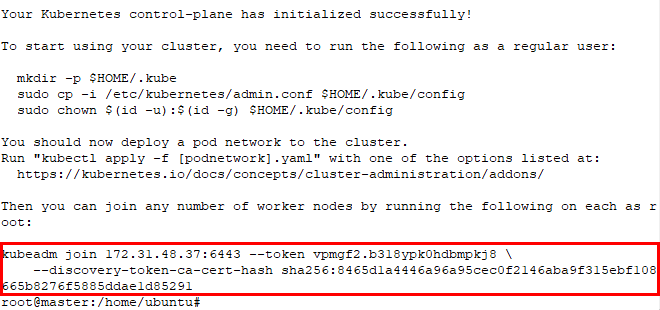
1. Start kubeadm **only on master.**

$ sudo su

# kubeadm init



If it runs successfully, it will provide a join command which can be used to join the master. Make a note of the highlighted part.

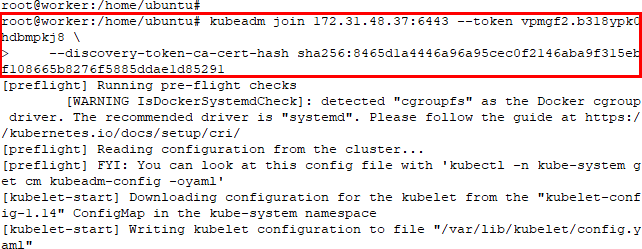


## Task 4: Joining a Cluster

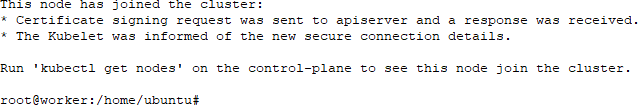
1. Run the **kubeadm join** command in **slave node,** that was previously noted from the

**master node** in the previous task.

# kubeadm join --token <your\_token> --discovery-token-ca-cert- hash <your\_discovery\_token>



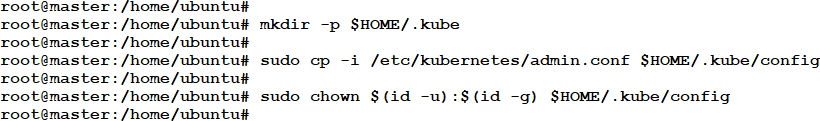
If node join was successful, **Node join complete** will be returned as shown below:



1. Run the following commands to configure **kubectl** on **master.**

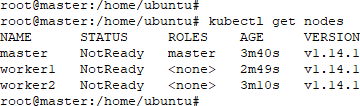
# mkdir -p $HOME/.kube

# sudo cp -i /etc/kubernetes/admin.conf $HOME/.kube/config # sudo chown $(id -u):$(id -g) $HOME/.kube/config



1. View node information on the **master**. The nodes will not be ready.

# kubectl get nodes

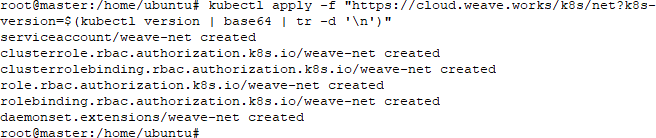


## Task 5: Deploy Container Networking Interface

In this task, CNI provided by WeaveWorks will be deployed in the **Master node.**

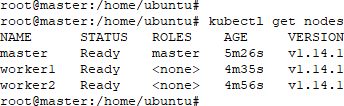
1. Apply weave CNI (Container Network Interface) as shown below:

# kubectl apply -f "https://cloud.weave.works/k8s/net?k8s- version=$(kubectl version | base64 | tr -d '\n')"



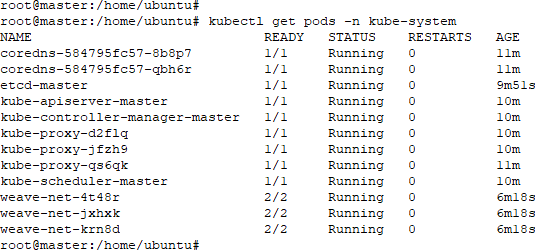
1. View nodes to see that they are **ready.**

# kubectl get nodes



1. View all Pods including system Pods and see that dns and weave are running.

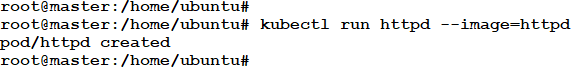
# kubectl get pod -n kube-system



## Task 6: Create Pods

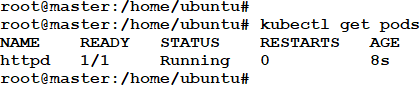
1. Create a Pod called **http** based on a Docker image on the **master.**

# kubectl run httpd --image=httpd



1. View the status of Pod and make sure it’s in the **running** state.

# kubectl get pods



1. Get a shell to the container in the Pod using the Pod name from the previous step.

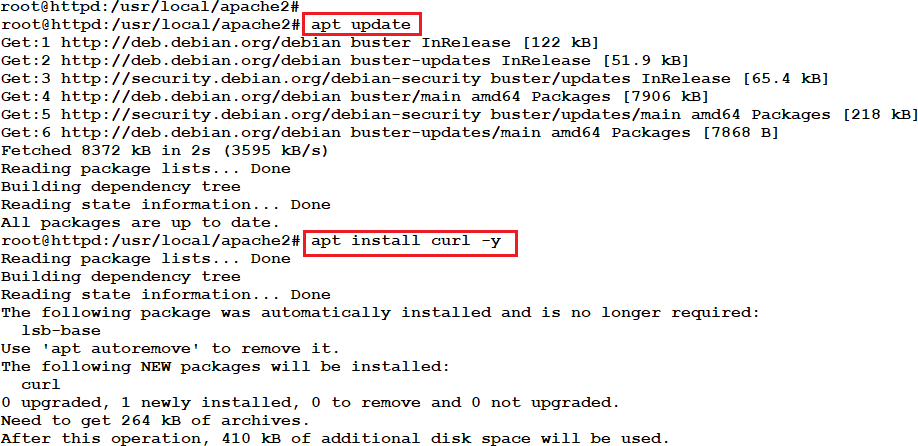
# kubectl exec -it <pod\_name> -- /bin/bash



1. Install curl in the container.

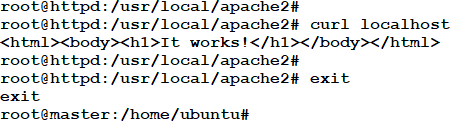
# apt update

# apt install curl -y



1. Run curl on the localhost (container) to verify the http installation.

# curl localhost # exit



# Introduction to Jobs and CronJobs in Kubernetes

Jobs are the controllers that supervise Pods carrying out specific tasks. Jobs ensure that the pods execute their commands, and tasks reach the completion stage.

Once a job is scheduled, the controller on the master node will wait until the pod is successfully terminated, which means containers will return an exit code 0. Once the job is done, the container will take the pod off the radar. After the job is completed, no more pods are created but the existing ones are not deleted either. Such pods can be used to acquire logs to check for any errors, warnings, or other diagnostic outputs.

It’s up to the user to delete old jobs, after checking their status.

There are three main types of tasks, which are suitable to run as a Job:

* Non-parallel jobs
* Parallel jobs with a fixed number of completions
* Parallel jobs with a work queue

**Non-parallel Jobs**

* It only runs 1 Pod.
* It creates a replacement if the pod goes down.
* Non-parallel jobs are completed when the pod terminates successfully after the operation.

**Parallel Jobs**

* With a fixed completion count: We can specify the number of Pods needed to complete the Job wherein multiple pods run parallelly.

     Use cases: Send e-mails, rendering frames, transcoding files, scanning ranges of Keys in a NoSQL database, etc.

* With a work queue: Pods are created, and they use a queue to pick up tasks one after the other for processing. The process is repeated one after the other until all the tasks in the queue are finished.

**Use Cases**:

* A storage service needs to be started to hold the work queue.
* Fill a queue with messages after creating it.
* Start a Job that starts several Jobs

**CronJob**

CronJobs in Kubernetes are like the ones that we hear while working with Unix/Linux platforms. They come under the scheduled jobs category, and as the name suggests, we can schedule the timing of such jobs. For example, you wish to run a Pod every hour, accumulate the logs from the system, zip it, and push it to your storage like AWS S3 or Google Cloud Storage. Later, all log files on the disk need to be deleted as well. CronJobs help you achieve such tasks.

Let’s discuss about CronJobs below:

* CronJob creates Jobs on a time-based schedule
* Similar to crontab on Unix/Linux
* Use case:
* Creating periodic and recurring tasks, like running backups or sending e-mails.
* Used to schedule individual tasks at a specific time, such as, we can have a job that checks our system’s capacity and send us an e-mail if it’s above 90%.

**Parameters**

* startingDeadlineSeconds: defines the deadline in seconds for starting the job if it misses its scheduled time for any reason.
* concurrencyPolicy: defines how the concurrent executions of the jobs created by this, will be treated.
* allow (default), forbid, replace.
* successfulJobsHistoryLimit : defines how many completed jobs should be kept in history.

# Lab 2: Jobs and Cronjobs in Kubernetes

## Task 1: Jobs

1. Create a yaml file called jobs.yml. use the content given below.

# vi jobs.yml



apiVersion: batch/v1

kind: Job

metadata:

name: jobs-hello

spec:

template:

metadata:

name: jobs-pod

spec:

containers:

- name: jobs-ctr

image: busybox

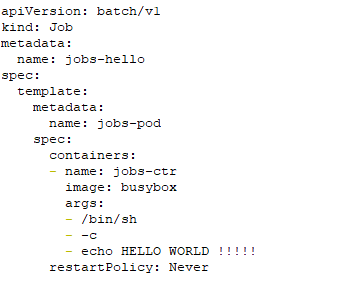
args:

- /bin/sh

- -c

- echo HELLO WORLD !!!!!

restartPolicy: Never



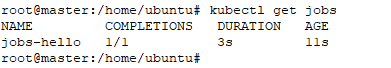
1. Create the job by applying the above yml file.

# kubectl create -f jobs.yml



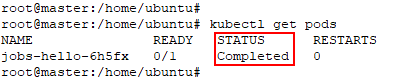
1. To print the list of all jobs in the cluster, use the following command.

# kubectl get jobs



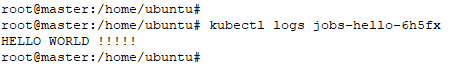
1. To verify the job task has run successfully we will check the status of pods.

# kubectl get pods



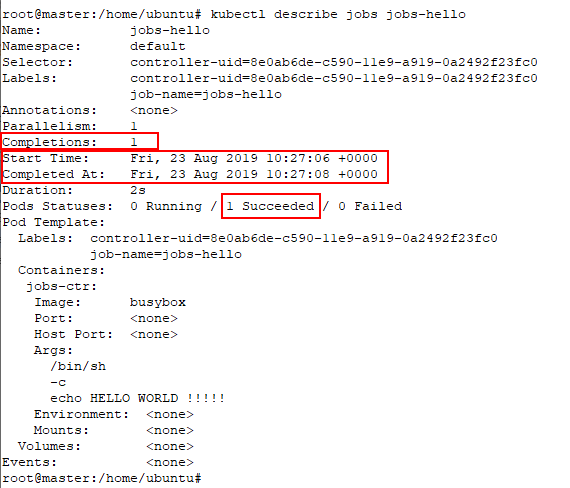
1. The pod has executed successfully, the output of the pod can be seen in the logs.

# kubectl logs **<pod\_name>**



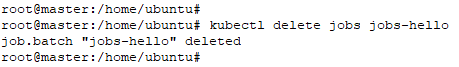
1. To get the complete details of the job and troubleshoot, use the following command.

# kubectl describe jobs **<job\_name>**



1. Delete the job.

# kubectl delete jobs **<job\_name>**



## Task 2: Cronjobs

1. Create a yaml called cron.yaml. Use the content given below to fill the file.

# vi jobs-cron.yml



apiVersion: batch/v1beta1

kind: CronJob

metadata:

name: cronjob-hello

spec:

schedule: "\*/1 \* \* \* \*"

jobTemplate:

spec:

template:

spec:

containers:

- name: cronjob-ctr

image: busybox

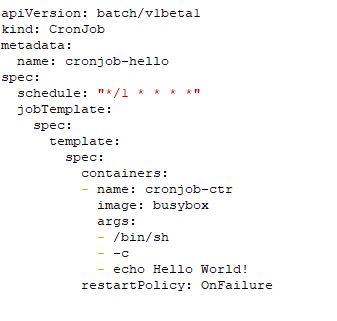
args:

- /bin/sh

- -c

- echo Hello World!

restartPolicy: OnFailure



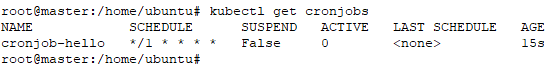
1. Create the cronjob using the yaml from the previous step.

# kubectl create -f jobs-cron.yml

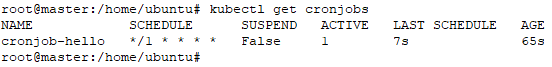


1. View the status of the cronjob.

# kubectl get cronjobs



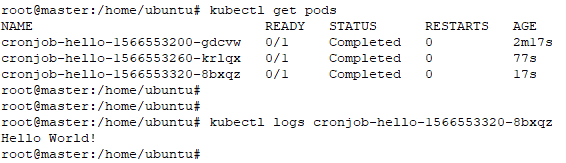
Wait for about a minute and run the same command again and notice that the **LAST SCHEDULE** field is now populated.



1. View the pods created by the job. Use the pod name to view its logs and verify that the jobs have run successfully.

# kubectl get pods

# kubectl logs **<pod\_name>**



1. Delete the cronjob.

# kubectl delete -f jobs-cron.yml





1. Create the cronjob using the yaml from the previous step.

$ kubectl create -f cron.yaml

Text, letter

Description automatically generated

1. View the status of the cronjob.

$ kubectl get cronjobs



Wait for about a minute and run the same command again and notice that the **LAST SCHEDULE** field is now populated.

Text, letter

Description automatically generated with medium confidence

1. View the pods created by the job. Use the pod name to view its logs and verify that the jobs have run successfully.

$ kubectl get pods

$ kubectl logs <pod\_name>

Table

Description automatically generated

1. Delete the CronJob.

$ kubectl delete -f cron.yaml

# Introduction to Sidecar Container

**Sidecar Pattern**

In Kubernetes, a pod contains one or more containers which shares the same volume and network. A sidecar is a secondary container in a pod that is loosely coupled to the main application container.

Sidecar containers are often used to enhance or improve the functionality of the application container, without the application container’s knowledge.

The sidecar container shares the same storage and network as of the main application containers and shares the same lifecycle as of the main application.

Some of the common examples of sidecar containers are log shippers, log watchers, and monitoring agents.

A close up of a logo

Description automatically generated

**Ambassador Pattern**

In this case, we have a secondary container which can be used to proxy network connections from the main application container to other services. It’s designed this way so that each container can do its task well. If the pod receives other tasks requiring the application function, those tasks can be handed over to the sidecar container.

Example: If an application initiates a request to localhost:6380, it’s the responsibility of the Ambassador container to receive the request and relay it to the servers defined in its configuration.

**Adapter Pattern**

They transform the output of the main container, and this pattern is important in scenarios where we need to monitor the output of different applications. Since the exported data from each application will not be same, we need a monitoring system which can convert all the different types of data into single unified representation by creating Pods that groups the application containers with adapters that know how to do the transformation.

**Init Containers:**

In Kubernetes, an Init Container is a special type of container that runs and completes before the main application container is started. The primary purpose of an Init Container is to perform some initialization or setup tasks that need to be completed before the main application container can start.

Here are some key concepts and features of Init Containers in Kubernetes:

* Separate container: An Init Container is a separate container in the same pod as the main application container. The Init Container runs and completes before the main application container starts.
* Initialization tasks: Init Containers are typically used to perform initialization tasks that are required before the main application container can start. These tasks can include things like downloading data or dependencies, setting up a database, or initializing configuration data.
* Dependencies: Init Containers can be used to ensure that the dependencies required by the main application container are available before the application container starts. This can help ensure that the application starts successfully and avoids issues related to missing dependencies.
* Order of execution: Init Containers are executed in the order in which they are defined in the pod spec. If an Init Container fails, Kubernetes will attempt to restart it until it succeeds. If an Init Container fails repeatedly, Kubernetes may terminate the entire pod and try to recreate it.
* Logging: Init Containers have their own separate logging, which can be useful for debugging and troubleshooting issues related to initialization and setup.
* Kubernetes API: Init Containers can be defined using the Kubernetes API or using YAML files. The **initContainers** field in the pod spec is used to define the Init Containers.

Overall, Init Containers are a useful feature of Kubernetes that allow you to perform initialization and setup tasks before the main application container starts. By using Init Containers, you can ensure that your application starts successfully and avoid issues related to missing dependencies or initialization tasks.

# Lab 3: Sidecar container

In this lab, we are going to create the main application container which writes the current date to the log file every five seconds, and we are going to create a sidecar container as nginx server which serves that log file.

## Task:1 Creating an Application Container with Sidecar Container

1. Create file sidecar.yml.

$ vi sidecar.yml

apiVersion: v1

kind: Pod

metadata:

name: pod-sidecar

spec:

containers:

- name: app-container

image: ubuntu:latest

command: ["/bin/sh"]

args: ["-c","while true; do date >> /var/log/app.txt; sleep 5; done"]

volumeMounts:

- name: share-logs

mountPath: /var/log/

- name: sidecar-container

image: nginx:latest

ports:

- containerPort: 80

volumeMounts:

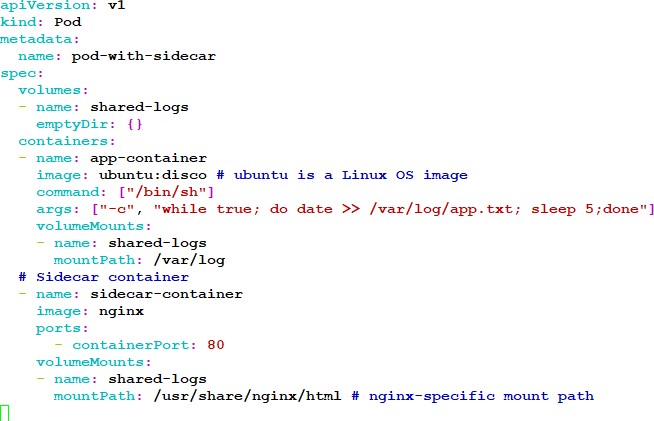
- name: share-logs

mountPath: /usr/share/nginx/html

volumes:

- name: share-logs

emptyDir: {}



1. Run the container by executing the following command.

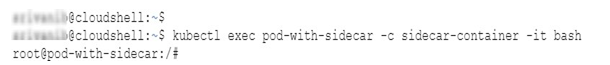
$ kubectl create -f sidecar.yml

Text, letter

Description automatically generated

1. Make sure that pod is created.

$ kubectl exec pod-with-sidecar -c sidecar-container -it bash



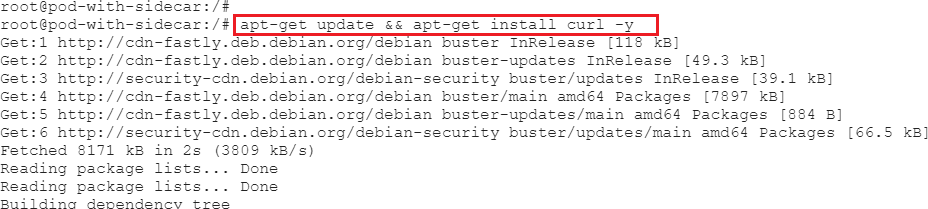
1. Connect to sidecar pod.

$ kubectl get pods



1. Update and Install curl on sidecar container.

# apt-get update && apt-get install curl -y



1. Access log file via nginx sidecar container server

# curl 'http://localhost:80/app.txt'

Table

Description automatically generated

1. Exit from the sidecar container and delete the pod.

# exit

$ kubectl delete -f sidecar.yml

Graphical user interface, text, application

Description automatically generated

# Deployment Strategies

In Kubernetes, a deployment strategy outlines how updates to your application's pods are managed. Different strategies cater to various requirements, balancing factors like downtime, reliability, and resource utilization. Here are some commonly used deployment strategies in Kubernetes:

## Rolling Update deployment

This is the default strategy. It gradually replaces old pods with new ones, ensuring a smooth transition. The number of new pods is increased while the old ones are scaled down, minimizing downtime and maintaining application availability.

## Recreate deployment

In a recreate deployment strategy, it terminates all existing pods and launches new ones with the updated version. It's useful when the old and new versions can't coexist. Downtime occurs during this process, making it suitable for applications with more flexible availability requirements.

## Blue/Green deployment

The blue-green deployment strategy, sometimes referred to as red-black deployment, involves a methodical approach to updating applications. In this strategy, the "blue" phase signifies the current application version, while the "green" phase represents the new version. Only one version is active and receiving traffic at any given time. During the deployment process, traffic is directed to the blue version while the green version is constructed and tested thoroughly. Upon successful testing, the traffic is then redirected to the new green version.

Once the deployment is deemed successful, there are a few options. The blue version can be retained as a backup for potential rollbacks or retired altogether. Alternatively, the blue environment can be used to deploy an even newer version of the application. In this scenario, the existing blue environment acts as a testing ground for upcoming releases.

This technique effectively eradicates downtime issues encountered in other deployment strategies like the recreate approach. Moreover, the blue-green deployment minimizes risks. If any unexpected complications arise with the new green version, a swift rollback to the previous blue version is possible. This provides an instant mechanism for both deploying and rolling back changes. Additionally, versioning problems are mitigated, as the entire application state undergoes a single update during the deployment process.

## Canary deployment

Canary deployment is a deployment strategy that enables you to roll out new versions of an application or service to a subset of users or traffic. In Kubernetes, canary deployment involves deploying a new version of an application to a small percentage of users or traffic while keeping the previous version running for the remaining users or traffic. The idea behind canary deployment is to gradually roll out changes, monitor the new version's performance, and gradually increase the percentage of users or traffic using the new version while gradually phasing out the previous version.

To implement a canary deployment in Kubernetes, you can use various deployment strategies, such as using deployment objects with labels and selectors to manage and control the rollout process. One common way to do this is by using a technique called "traffic splitting," which involves using a Kubernetes feature called "service mesh" to split traffic between two versions of the same application.

In this approach, you create two deployments with different versions of the application and then use a service mesh like Istio to split traffic between them. You can start by routing, for example, 10% of the traffic to the new version and the remaining 90% to the old version. You can then monitor the new version's performance, including metrics such as latency, errors, and user feedback. If the new version is performing well, you can gradually increase the traffic to it while decreasing traffic to the old version.

Canary deployment allows you to roll out changes to your application or service gradually, reducing the risk of downtime or service disruption while allowing you to test new features or fixes. It is an effective way to test new features, assess performance, and minimize the impact of changes on users or customers.

# Lab 4 Canary Deployment on Kubernetes

1. Create a deployment using below yaml to deploy pods for our web-blue app.

vi web-blue.yaml

apiVersion: apps/v1

kind: Deployment

metadata:

name: web-blue

spec:

replicas: 3

selector:

matchLabels:

app: web-blue

type: web-app

strategy:

type: RollingUpdate

template:

metadata:

labels:

app: web-blue

type: web-app

spec:

containers:

- image: mandarct/web-blue:v1

name: web-blue

ports:

- containerPort: 80

protocol: TCP

A screenshot of a computer

Description automatically generated with medium confidence

1. Deploy the above deployment to the Kubernetes cluster in the default namespace.

# kubectl apply -f web-blue.yaml

1. Verify that pods are running.

# kubectl get po

NAME READY STATUS RESTARTS AGE

web-blue-5657b94c87-cqkfz 1/1 Running 0 12m

web-blue-5657b94c87-rwcfj 1/1 Running 0 12m

web-blue-5657b94c87-vgsqv 1/1 Running 0 12m

Schematic

Description automatically generated with medium confidence

1. Create a service of type Load-balancer to expose above deployment using the following yaml.

vi svc-web-lb.yaml

apiVersion: v1

kind: Service

metadata:

name: web-app-svc-lb

spec:

ports:

- port: 80

protocol: TCP

targetPort: 80

selector:

type: web-app

type: LoadBalancer

ports:

- port: 80

targetPort: 80

1. Deploy this Load-Balancer service to the default namespace.

kubectl apply -f svc-web-lb.yaml

1. Verify the service is created of type load-balancer.

kubectl get svc web-app-svc-lb

NAME TYPE CLUSTER-IP EXTERNAL-IP PORT(S) AGE

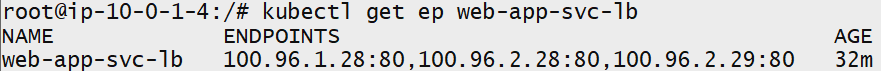
web-app-svc-lb LoadBalancer 100.67.144.247 a72cd7c5e674044e4b09e34ae1848acd-702623717.ap-south-1.elb.amazonaws.com 80:30229/TCP 18m

1. Verify the end-point object is created pointing to the IP address for web-blue pods.

kubectl get ep web-app-svc-lb

NAME ENDPOINTS AGE

web-app-svc-lb 100.96.1.28:80,100.96.2.28:80,100.96.2.29:80 32m



* Search for a72cd7c5e674044e4b09e34ae1848acd-702623717.ap-south-1.elb.amazonaws.com in **AWS -> EC2 Dashboard -> Load balancers**
* Verify that a new **ELB** has been created in AWS. Wait for 2 minutes for the ELB instances to be in-service.

Graphical user interface, text, application, email

Description automatically generated

1. Test the ELB DNS URL from your browser, you should get below response from the web-blue app pods.

Graphical user interface, text, application, email

Description automatically generated

1. Create another deployment using the following yaml.

vi web-green.yaml

apiVersion: apps/v1

kind: Deployment

metadata:

name: web-green

spec:

replicas: 3

selector:

matchLabels:

app: web-green

strategy:

type: RollingUpdate

template:

metadata:

labels:

app: web-green

type: web-app

spec:

containers:

- image: mandarct/web-green:v1

name: web-green

ports:

- containerPort: 80

protocol: TCP

1. Deploy the above deployment to the Kubernetes cluster in the default namespace.

kubectl apply -f web-green.yaml

1. Verify pods running for web-green deployment as well.

kubectl get po

NAME READY STATUS RESTARTS AGE

web-blue-5657b94c87-cqkfz 1/1 Running 0 25m

web-blue-5657b94c87-rwcfj 1/1 Running 0 25m

web-blue-5657b94c87-vgsqv 1/1 Running 0 25m

web-green-76df95dbcd-4bnkf 1/1 Running 0 27m

web-green-76df95dbcd-57v5x 1/1 Running 0 27m

web-green-76df95dbcd-rhmvk 1/1 Running 0 27m

Text, table

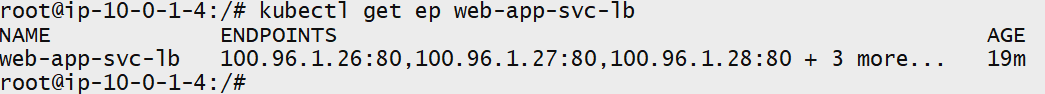
Description automatically generated

1. Verify that the endpoints for the existing load-balancer service are updated with pods for web-green deployment.

root@ip-10-0-1-4:/# kubectl get ep web-app-svc-lb

NAME ENDPOINTS AGE

web-app-svc-lb 100.96.1.26:80,100.96.1.27:80,100.96.1.28:80 + 3 more... 19m



1. Hit the load balancer (ELB) URL from web-browser, multiple times. You should be below 2 outputs, as the traffic is routed between the 2 deployments (web-blue & web-green).

Graphical user interface, application

Description automatically generated

Graphical user interface, text, application

Description automatically generated

1. Once we have deployed both blue and green versions of our deployments, we notice that pods are created with below labels. Our Load balancer service is created with matching labels for ‘type=web-app’, so the traffic is distributed (load balanced) across both the versions of our deployments.

A picture containing graphical user interface

Description automatically generated

1. If you delete the web-green deployment, load-balancer will start sending traffic only to the blue pods.

**kubectl delete deploy web-green**

deployment.apps "web-green" deleted

Icon

Description automatically generated with low confidence

1. The end point object for load balancer service will be back pointing only to the IP address for web-blue pods.

kubectl get ep web-app-svc-lb

NAME ENDPOINTS AGE

web-app-svc-lb 100.96.1.28:80,100.96.2.28:80,100.96.2.29:80 32m

A picture containing text, orange, close

Description automatically generated

1. Same can be verified by describing the service.

kubectl describe svc web-app-svc-lb

Name: web-app-svc-lb

Namespace: default

Labels: <none>

Annotations: kubectl.kubernetes.io/last-applied-configuration:

{"apiVersion":"v1","kind":"Service","metadata":{"annotations":{},"name":"web-app-svc-lb","namespace":"default"},"spec":{"ports":[{"port":8...

Selector: type=web-app

Type: LoadBalancer

IP: 100.67.144.247

LoadBalancer Ingress: a72cd7c5e674044e4b09e34ae1848acd-702623717.ap-south-1.elb.amazonaws.com

Port: <unset> 80/TCP

TargetPort: 80/TCP

NodePort: <unset> 30229/TCP

Endpoints: 100.96.1.28:80,100.96.2.28:80,100.96.2.29:80

Session Affinity: None

External Traffic Policy: Cluster

Events:

Type Reason Age From Message

---- ------ ---- ---- -------

Normal EnsuringLoadBalancer 38m service-controller Ensuring load balancer

Normal EnsuredLoadBalancer 38m service-controller Ensured load balancer

1. Try hitting the ELB URL from web-browser multiple times, you should only see the response from web-app-blue.

Graphical user interface, application

Description automatically generated

# Liveness, Readiness and Startup Probes

A Microservice-based application is a methodology that structures the application as a collection of services that perform the task efficiently. These microservices are highly available, loosely coupled, and independent.

Along with the benefits, it also has some challenges. In microservice-based applications, all the services must work for the application to function. If there is a break, the system must automatically detect, perform a root cause analysis and fix them.

Health checks let the system know if the instances of your application are working or not. If an instance is not working, it must be in idle state and other services should not access it or send a request to it. Instead, requests are sent to another instance of the application that is up and ready. At the same time, the liveness probe must get the idle machine back to a healthy state.

The probe is a diagnostic performed periodically by the kubelet on a Container.

Kubelet calls the handler implemented by the container to perform the diagnostic.

There are three types of handlers:

* **ExecAction**: A specified command is executed inside the container. The diagnostic exits with a status code 0 if the command is successful.

* **TCPSocketAction**: A TCP check is performed against the Container’s IP address on a specified port. The diagnostic is successful if the port is open.
* **HTTPGetAction**: A HTTP Get request is performed against the Container’s IP address on a specified port and path. The diagnostic is successful if the response has a status code greater than or equal to 200 and less than 400

By default, Kubernetes sends traffic to a pod when all the containers inside the pod are started or restarted. We can also create custom health checks that suites our application.

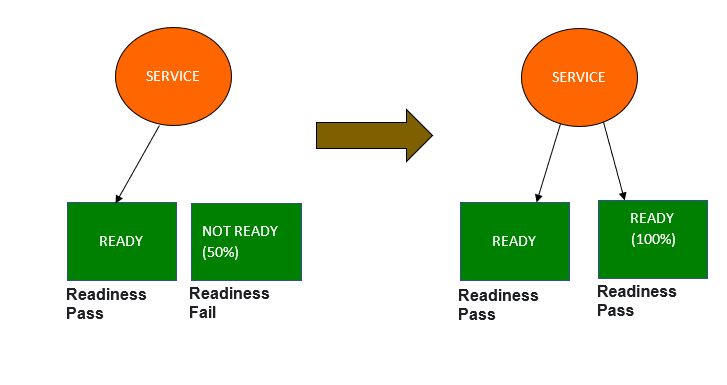
Kubernetes provides three types of health checks:

**Readiness probe**

Readiness probes are used to decide when a container is ready to accept the traffic. A Pod is ready when all its containers are ready. Kubernetes ensures readiness probe to be passed before allowing a service to send traffic to the pod.

Use Cases:

1. Readiness probes can be used to control which Pods are used as backends for Services. If a readiness fails, Kubernetes does not forward traffic to the pod until the container is ready.
2. Some applications might take time to start up, but Kubernetes starts sending traffic as soon as containers are running. In this case, readiness probes can be used to ensures the application is running before service is sending traffic to it.



**Liveness probe**

Liveness probes allow Kubernetes to know if your application is alive or dead. If it is, alive, then Kubelet leaves it alone. If it is dead, Kubernetes removes the Pod and replaces it with a new one. It tells Kubernetes when to restart the container.

For example, a liveness probe could detect a deadlock and unable to make progress. Restarting the container will make the application available.



Liveness and Readiness probes are used with Kubernetes deployments to achieve:

• Enable zero-downtime deploys

• Ensure that failed containers are automatically restarted

**Startup Probe**

As evident from its name, Startup probes let Kubelet know when a container application has started. One interesting thing to note here is, if Startup probe is configured, it makes sure that the other two probes, Liveness and Readiness probes are disabled until the checks by Startup probes are completed. This setting is helpful to avoid any kind of interference from other probes while startup probe is in action. Startup probes can be used to avoid slow starting containers getting killed by the kubelet.

**Probing parameters**

* initialDelaySeconds: waiting time before sending a probe after a container starts.
* timeoutSeconds: Time it takes to respond before a request is considered a failure.
* periodSeconds: How often a probe will be sent.
* successThreshold: Minimum consecutive successes for the probe to be considered successful after having failed.
* failureThreshold: When a Pod starts and the probe fails, Kubernetes will try failureThreshold times before giving up.

**Probe use case:**

* When updating deployments, Kubernetes waits for a replacement pod to start running before removing the old pod. And if a pod stops running, it also automatically tries to restart the pod.
* These probes prove their importance in between the time when a pod starts running and when your service starts functioning.
* Kubernetes shows us the running status of a pod, probes helps us to know if your container is functioning or not.

**AutoHealing procedure**

* Readiness probe fails
* Kubernetes stops routing traffic to the pod
* Liveness probe fails
* Kubernetes restarts the pod
* Readiness probe succeeds
* Kubernetes starts routing traffic to the pod again

# Lab 5: Liveness and Readiness probes

This lab demonstrates the functionality of Liveness and Readiness probes.

## Task 1: Liveness probes

1. Create a Pod yaml and enter the content given below.

$ vi liveness-pod.yaml



apiVersion: v1

kind: Pod

metadata:

labels:

app: lns

name: liveness-pod

spec:

containers:

- name: liveness

image: busybox

args:

- /bin/sh

- -c

- touch /liveness; sleep 6000

livenessProbe:

exec:

command:

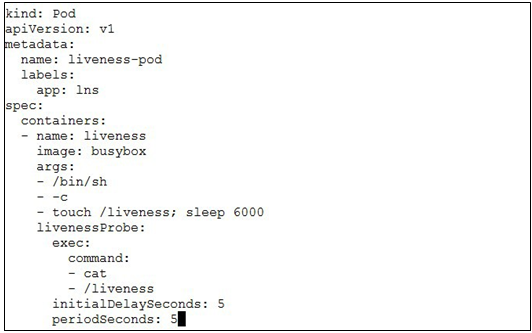
- cat

- /liveness

initialDelaySeconds: 5

periodSeconds: 5

Once entered, it should look as shown below.



1. Apply the created pod yaml.

$ kubectl create -f liveness-pod.yaml

1. View that the Pod is running with no restarts.



Text, letter

Description automatically generated

1. Get shell access to the container and delete the **/liveness** file.

$ kubectl exec -it liveness-pod sh

# rm -f /liveness

# exit

Graphical user interface, text, application

Description automatically generated

1. View container events and see that the liveness probe has failed.

$ kubectl describe pod liveness-pod



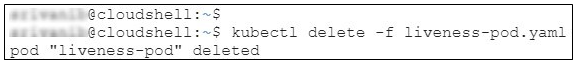
1. View container status and see that it has restarted due to failed liveness probe.

Text, table

Description automatically generated

1. Delete all the resources created in this task.

$ kubectl delete -f liveness-pod.yaml



## Task 2: Readiness probe

1. Create a Pod yaml and enter the contents given below.

$ vi readiness-pod.yaml

Once entered, it should look as shown below.



apiVersion: v1

kind: Pod

metadata:

labels:

app: rns

name: readiness-pod

spec:

containers:

- name: readiness

image: nginx

args:

- /bin/bash

- -c

- service nginx start; touch /readiness; sleep 6000

readinessProbe:

exec:

command:

- cat

- /readiness

initialDelaySeconds: 5

periodSeconds: 5

Text

Description automatically generated

1. Create a service yaml and enter the contents given below.

$ vi readiness-svc.yaml

A picture containing text

Description automatically generated

Once entered, it should look as shown below.

kind: Service apiVersion: v1 metadata:

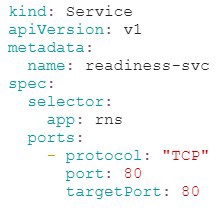
name: readiness-svc spec:

selector: app: rns

ports:

- protocol: "TCP" port: 80

targetPort: 80



1. Apply the pod and service yamls.

$ kubectl create -f readiness-pod.yaml

$ kubectl create -f readiness-svc.yaml

Table

Description automatically generated

1. Describe the service and notice that endpoints is populated with the Pod’s IP.

$ kubectl describe svc readiness-svc

Text

Description automatically generated

1. Get shell access to the container and delete the **/readiness** file.

$ kubectl exec -it readiness-pod bash

# rm -f /readiness

# exit

Graphical user interface, text, application

Description automatically generated

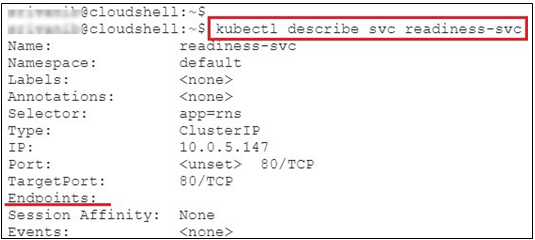
1. View the Pod events and see that readiness probe has failed.

$ kubectl describe pods readiness-pod



1. Describe service and see that the endpoints field it not populated meaning the traffic is not being routed to the Pod.

$ kubectl describe svc readiness-svc



1. Delete all the resources created in this task.

$ kubectl delete -f readiness-pod.yaml

$ kubectl delete -f readiness-svc.yaml

Table

Description automatically generated

# Concept of Monitoring and Logging

Monitoring in Kubernetes is the process of collecting, analysing, and visualizing various metrics and logs from different components of the cluster to gain insights into the health, performance, and availability of the applications running on the cluster. Kubernetes provides several built-in monitoring tools and integrations with third-party monitoring solutions that help you monitor your cluster and applications.

The monitoring in Kubernetes typically involves the following components:

1. **Metrics API:** Kubernetes provides a Metrics API that exposes the performance metrics of various Kubernetes components, including pods, nodes, and containers. These metrics can be used for real-time monitoring and alerting.
2. **Prometheus:** Prometheus is a popular open-source monitoring solution that can be integrated with Kubernetes to monitor the performance of applications running on the cluster. It collects metrics from different components of the cluster and provides a powerful query language to analyse and visualize the data.
3. **Grafana:** Grafana is an open-source visualization platform that can be used to create dashboards and alerts based on the metrics collected by Prometheus. It provides a wide range of visualization options and supports various data sources.
4. **Kubernetes Dashboard:** Kubernetes Dashboard is a web-based user interface that provides a graphical representation of the cluster and allows you to monitor various Kubernetes resources such as pods, deployments, and services.
5. **Logging:** Kubernetes provides various logging mechanisms, such as container logging, node-level logging, and cluster-level logging, that allow you to collect and analyse logs from different components of the cluster.

To effectively monitor your Kubernetes cluster, you need to define monitoring requirements and choose appropriate monitoring solutions that meet those requirements. Some of the key factors to consider when choosing a monitoring solution include ease of installation and configuration, scalability, support for different data sources and data types, query performance, and alerting capabilities.

In summary, monitoring is a crucial aspect of running Kubernetes clusters as it enables you to proactively identify and address issues that can impact the availability and performance of your applications. By leveraging the various monitoring tools and integrations provided by Kubernetes, you can gain deep insights into the health and performance of your cluster and ensure optimal operation of your applications.

Logging

Logging in Kubernetes is the process of collecting and analysing log data generated by the various components of a Kubernetes cluster, including containers, pods, nodes, and the Kubernetes API server itself. This log data can be used to gain insights into the health, performance, and availability of the cluster and its applications.

There are several approaches to logging in Kubernetes, including:

1. **Container-level logging**: Kubernetes allows you to collect logs from containers running inside pods by redirecting the stdout and stderr streams to a logging driver. The most common logging drivers used in Kubernetes are JSON-file, journald, and syslog.
2. **Node-level logging**: Kubernetes also provides support for node-level logging, which involves collecting logs from system components running on the nodes, such as the kubelet and container runtime. Popular logging solutions for node-level logging include Fluentd and Fluent Bit.
3. **Cluster-level logging:** Kubernetes also supports collecting logs at the cluster level, which involves collecting logs from the Kubernetes API server and other control plane components. Solutions such as Elasticsearch, Logstash, and Kibana (ELK stack) and Grafana Loki are commonly used for cluster-level logging.

To effectively collect and analyze log data in Kubernetes, you need to consider the following:

1. Logging drivers: You need to choose the appropriate logging driver that best suits your requirements, taking into account factors such as log format, storage location, and retention policy.
2. Log aggregation: You need to choose a log aggregation solution that can collect and centralize log data from various sources, provide search and filtering capabilities, and support visualization and analysis.
3. Storage and retention: You need to choose a storage solution that can handle the volume of log data generated by your applications and provide appropriate retention policies based on your requirements.
4. Alerting: You need to set up alerts based on log data to detect and respond to issues in real-time.

Logging in Kubernetes is a critical aspect of maintaining the health and performance of your cluster and applications. By effectively collecting and analyzing log data, you can gain insights into the behavior of your applications, troubleshoot issues, and improve the overall performance and availability of your cluster.

LAB 6 : Monitoring and Logging

The Metrics Server in Kubernetes is like a resource tracker that keeps an eye on how much CPU and memory is being used in the entire cluster. It gathers data about resource usage and then displays statistics about it. This helps you keep track of how much resources are used by each computer (node) and each task (pod) in your cluster

## Task 1: Setting Metric Server in Kubernetes

1. Install metric server by using below command

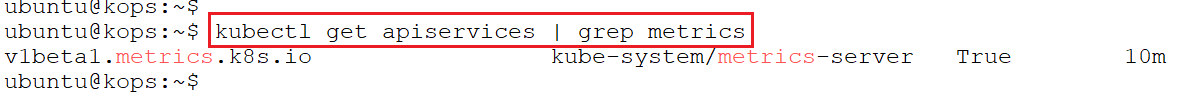
$ kubectl apply -f https://github.com/kubernetes-sigs/metrics-server/releases/latest/download/components.yaml

A close-up of a document

Description automatically generated

1. You can check whether the metrics server is installed correctly or not by using the following commands:

$ kubectl get apiservices | grep metrics



$ kubectl -n kube-system get po

A screenshot of a computer program

Description automatically generated

Task 2: Monitor Resources by using Metric Server

1. Use the following command to show the resource consumption details for each node in your cluster, including information about how much CPU and memory is being used

$ kubectl top node

A close-up of a computer code

Description automatically generated

1. To observe the resource usage of pods within your current namespace, use the following command:

$ kubectl top pod

A close-up of a text

Description automatically generated

1. To observe the resource usage of a specific pods , use the following command:

$ kubectl top pod -l run=httpd

A close-up of a computer code

Description automatically generated

1. To observe the CPU and Memory usage of pods within a specific namespace, use the following command:

$ kubectl top pod -A –sort-by=cpu

A screenshot of a computer program

Description automatically generated

$ kubectl top pod –all-namespaces –sort-by=memory

A screenshot of a computer

Description automatically generated

# ConfigMaps and Secrets

In Kubernetes, ConfigMap and Secret are two types of objects that allow you to store and manage configuration data and sensitive information, respectively, separately from your application code.

**ConfigMap:**

A ConfigMap is a Kubernetes object that stores configuration data as key-value pairs. ConfigMaps can be used to store configuration data such as environment variables, command-line arguments, and configuration files. ConfigMaps allow you to separate your application configuration from your application code, making it easier to manage and update your configuration data separately.

ConfigMaps can be created using YAML files, and they can be accessed by pods as environment variables or as mounted volumes. To access a ConfigMap as an environment variable, you can define the key-value pairs in the pod spec using the **env** field. To access a ConfigMap as a mounted volume, you can define a volume mount in the pod spec and specify the ConfigMap as the source of the volume.

**Secret:**

A Secret is a Kubernetes object that stores sensitive information, such as passwords, API keys, and certificates, separately from your application code. Secrets are stored in an encoded or encrypted format to protect the data from unauthorized access. Secrets can be used to store any data that should not be included in your application code or configuration files.

Like ConfigMaps, Secrets can be created using YAML files, and they can be accessed by pods as environment variables or as mounted volumes. To access a Secret as an environment variable, you can define the key-value pairs in the pod spec using the **env** field, and the value will be automatically decoded by Kubernetes. To access a Secret as a mounted volume, you can define a volume mount in the pod spec and specify the Secret as the source of the volume.

One important thing to note is that ConfigMaps and Secrets are not encrypted by default, and their contents are stored in plain text in etcd, the key-value store used by Kubernetes. If you need to store sensitive information in a Secret, you should use encryption or a third-party tool to protect the data.

Overall, ConfigMaps and Secrets are useful Kubernetes objects that allow you to store and manage configuration data and sensitive information separately from your application code. By using ConfigMaps and Secrets, you can simplify the management of your configuration data and improve the security of your applications.

# Lab 7: ConfigMap and Secret

## Task 1: Setting container environment variables using configmap

1. Create a ConfigMap yaml and enter the contents given below.

$ vi config-map.yaml



Once entered, it should look as shown below.

apiVersion: v1 kind: ConfigMap metadata:

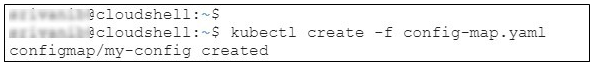
name: my-config namespace: default

data:

mydata: hello\_world

1. Create the ConfigMap using the yaml.

$ kubectl create -f config-map.yaml



Text

Description automatically generated with medium confidence

1. Create a pod that uses the ConfigMap using the content given below.

$ vi cm-pod.yaml



apiVersion: v1

kind: Pod

metadata:

name: cm-pod

spec:

containers:

- name: nginx

image: nginx

ports:

- containerPort: 80

env:

- name: hlo

valueFrom:

configMapKeyRef:

name: my-config

key: mydata

Once entered, it should look as shown below.

Text

Description automatically generated

1. Create the Pod using the yaml.

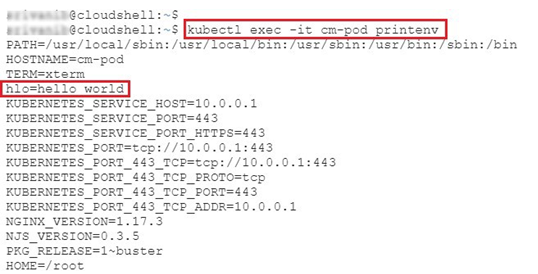
$ kubectl create -f cm-pod.yaml

Graphical user interface, text

Description automatically generated

1. Once the Pod is up, see that the environment variable specified in the ConfigMap is set in the container.

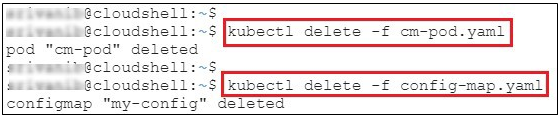
$ kubectl exec -it cm-pod printenv



1. Delete the resources created in this task.

$ kubectl delete -f cm-pod.yaml

$ kubectl delete -f config-map.yaml

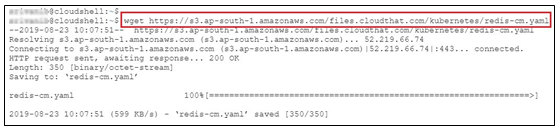


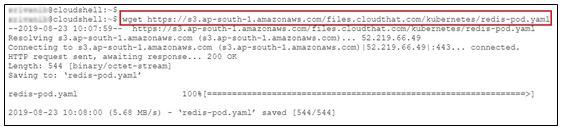
## Task 2: Setting configuration file with volume using ConfigMap

1. Download the ConfigMap and Pod yaml using the links.

**$ wget** [**http://files.cloudthat.com/kubernetes/redis-cm.yaml**](http://files.cloudthat.com/kubernetes/redis-cm.yaml)

**$ wget** [**http://files.cloudthat.com/kubernetes/redis-pod.yaml**](http://files.cloudthat.com/kubernetes/redis-pod.yaml)

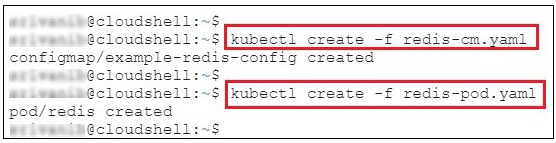




1. Create the resources using the yamls downloaded in the previous step.

$ kubectl create -f redis-cm.yaml

$ kubectl create -f redis-pod.yaml



1. See that the config file **redis.conf** is present at **/redis-master/** and is having the contents specified in the ConfigMap.

$ kubectl exec -it redis cat /redis-master/redis.conf



1. Delete all the resources created in this task.

$ kubectl delete -f redis-pod.yaml

$ kubectl delete -f redis-cm.yaml

Graphical user interface, text, application

Description automatically generated

## Task 3: Creating a Secret

1. Download the secrets file using command:

$ wget <http://files.cloudthat.com/kubernetes/lab6/secrets.yaml>

Graphical user interface, text, application

Description automatically generated

1. Apply the file to add the secret.

$ kubectl create -f secrets.yaml

Text, letter

Description automatically generated

1. Verify if the secret has been added.

$ kubectl get secrets

Text

Description automatically generated with low confidence

Resource Quota

In Kubernetes, a resource quota is a way to limit the amount of CPU, memory, and other resources that a group of pods or namespaces can use in a cluster. Resource quotas provide a way to enforce resource limits and prevent resource contention, which can lead to performance issues and service disruptions.

Here are some key concepts and features of resource quotas in Kubernetes:

* Resource quota objects: In Kubernetes, resource quotas are defined using YAML files that contain resource quota objects. Resource quota objects specify the limits on the amount of CPU, memory, and other resources that a group of pods or namespaces can use.
* Quota scopes: Resource quotas can be defined for different scopes, including a single namespace, multiple namespaces, or the entire cluster. This allows you to apply resource limits to specific groups of pods or namespaces, or to the entire cluster.
* Resource limits: Resource quotas define limits on the amount of CPU, memory, and other resources that can be used by pods in a group or namespace. These limits can be defined as absolute values or as a percentage of the total cluster resources.
* Resource requests: Resource quotas can also enforce minimum resource requests for pods in a group or namespace. Resource requests specify the amount of CPU, memory, and other resources that a pod requires to run, and resource quotas can prevent pods from running if they do not meet these requirements.
* Quota enforcement: Resource quotas are enforced by Kubernetes, which monitors the usage of resources by pods and namespaces and prevents them from exceeding the defined limits. If a pod or namespace exceeds its resource quota, Kubernetes may terminate the offending pods or prevent new pods from starting.
* Quota management: Resource quotas can be managed using the Kubernetes API or command-line tools like kubectl. You can create, update, and delete resource quotas as needed to manage the resource usage of your pods and namespaces.

Overall, resource quotas are a powerful tool for managing and controlling the resource usage of pods and namespaces in a Kubernetes cluster. By defining limits on the amount of CPU, memory, and other resources that can be used, resource quotas help ensure that your cluster can run efficiently and reliably, even under heavy loads and resource contention.

# Lab 8: 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

A picture containing graphical user interface

Description automatically generated

1. Verify namespace creation.

$ kubectl get ns

Table

Description automatically generated

## 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

A picture containing graphical user interface

Description automatically generated

1. Verify resourcequota creation.

$ kubectl get resourcequota -n quotas

A picture containing text

Description automatically generated

## 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.

Timeline

Description automatically generated with medium confidence

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

$ kubectl create -f rq-pod.yaml

A picture containing graphical user interface

Description automatically generated

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

$ kubectl get resourcequota -n quotas -o yaml

Text

Description automatically generated with medium confidence

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

Text

Description automatically generated

## Task 4: Limiting Number of Pods

In this task, the functionality of resourcequotas for several 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.

Text

Description automatically generated

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

A picture containing graphical user interface

Description automatically generated

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

$ vi rq-pod.yaml

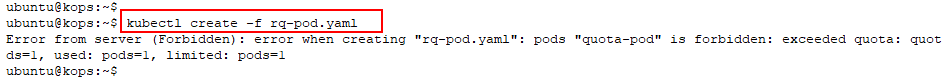


Timeline

Description automatically generated

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

Graphical user interface

Description automatically generated with medium confidence

# Security Context

A security context takes care of the operating system settings applied to a container. Some of the settings include User ID(UID), Group ID(GID), capabilities, SELinux role, etc. As you would expect, there are two levels of security context:

1. Pod level security context
2. Container level security context

**Pod level security context**:

Setting Pod level security context affects all the containers in the pod. Volume security policy is included in Pod level security context, and it handles volume level security settings as well, where applicable. Parameters of volume level security include fsGroup and seLinuxOptions.

**Container level security context**:

As the name suggests, container level security context affects only the specific container that we are working on. If there is an overlap between pod level and container level security context, the container level security context is given priority and taken into consideration. One interesting thing to note here is that, container level settings do not affect the pod’s volumes.

# Lab 9: Pod and Container level security using Context.

A security context defines permission to access an object, on pod and container level for security based on user ID (UID) and group ID (GID).

In this lab, we are going to create main application container which writes the current date to log file every five seconds and we are going to create sidecar container as nginx server which serves that log file.

## Task 1: Set the security context for a pod

1. Create file sc-pod.yml to apply security context on pod level. These security settings will be applied on all the containers in the pod.

# vi sc-pod.yml

apiVersion: v1

kind: Pod

metadata:

name: sc-pod

spec:

securityContext:

runAsUser: 1000

runAsGroup: 3000

fsGroup: 2000

volumes:

- name: sc-vol

emptyDir: {}

containers:

- name: sc-ctr

image: busybox

command:

- sh

- -c

- sleep 1h

volumeMounts:

- name: sc-vol

mountPath: /data/demo

securityContext:

allowPrivilegeEscalation: false

The yml file should look similar to the below image:

Text

Description automatically generated

1. Create the pod.

# kubectl create -f sc-pod.yml

Text

Description automatically generated

1. Verify that the pod’s containers are running.

# kubectl get pods

Text

Description automatically generated

1. Now SSH into it

# kubectl exec -it pod-sc -- sh

Text

Description automatically generated

When we go into the container, usually we will be the root user. We can identify that by looking into the shell prompt, “#”. But here the shell prompt is “$”, showing that we were not into the container as a root user.

1. List the running processes in the container. The output will show that all the processes are running as user 1000, i.e. **runAsUser.**

$ ps

A picture containing table

Description automatically generated

1. Navigate to /data and list the directories.

$ cd /data

$ ls -l

Text

Description automatically generated with medium confidence

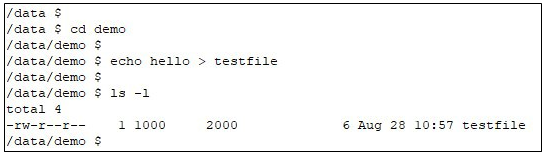
The output shows the directory, demo with the group id 2000 which is the value of **fsGroup.** The fsGroup owns the pod’s volumes, hence the directory belongs to the fsGroup.

1. Go into the demo directory and create a file. Then list the directory and notice the group ID that the file belongs to is 2000, i.e. **fsGroup.**

$ cd demo

$ echo hello > testfile

$ ls -l



1. Run the following command.

$ id



Exit the shell.

## Task 2: Set the security context for a container

The security settings that we specify for a container apply only to the container, and they override settings made at the Pod level when there is overlap. Containers setting do not affect the pod’s volumes.

1. Create file sc-ctr.yml to apply security context on container level.

# vi sc-ctr.yml

apiVersion: v1

kind: Pod

metadata:

name: sc-pod2

spec:

securityContext:

runAsUser: 1000

containers:

- name: sc-ctr2

image: gcr.io/google-samples/node-hello:1.0

securityContext:

runAsUser: 2000

allowPrivilegeEscalation: false

The yml file should look similar to the below image:

Graphical user interface, text, application, email

Description automatically generated

1. Create the pod.

# kubectl create -f sc-ctr.yml



1. Verify that the pod’s containers are running. Then SSH into it.

# kubectl get pods

# kubectl exec -it sc-pod2 -- sh

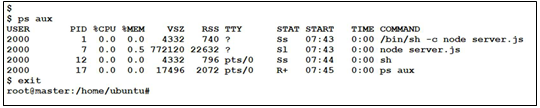
Text

Description automatically generated

1. List the running processes and **exit.**

$ ps aux

$ exit



The output will show that all the processes are running as user 2000, i.e. **runAsUser.** Notice that the value we specified for the pod was 1000. Hence, they override settings made at the pod level.

1. Delete the resources that we created in this lab.

# kubectl delete -f sc-pod.yml

# kubectl delete -f sc-ctr.yml

Text

Description automatically generated

# Network Policy

In Kubernetes, network policies are a way to control and secure network traffic between pods in a cluster. Network policies define rules that allow or deny traffic based on various criteria, such as the source and destination pods, IP addresses, ports, and protocols. Network policies are implemented by network plugins, and different network plugins may have different capabilities and syntax for defining policies.

Here are some key concepts and features of network policies in Kubernetes:

* Network policy objects: In Kubernetes, network policies are defined using YAML files that contain network policy objects. Network policy objects specify the rules for allowing or denying traffic between pods based on various selectors and rules.
* Pod selectors: Network policies use pod selectors to determine which pods to apply the policy to. Pod selectors can be based on labels or namespaces, and they allow you to define policies for specific groups of pods.
* Rules: Network policies define rules that allow or deny traffic based on various criteria, such as IP addresses, ports, and protocols. Rules can be defined for incoming and outgoing traffic, and they can be combined using logical operators like AND and OR.
* Default policies: By default, Kubernetes allows all traffic between pods in a cluster. Network policies allow you to define more restrictive policies that deny traffic by default and only allow traffic that meets specific criteria.
* Namespace isolation: Network policies can be used to isolate traffic between pods in different namespaces, providing an additional layer of security and control.
* Network plugins: Network policies are implemented by network plugins, which may have different capabilities and syntax for defining policies. Some popular network plugins for Kubernetes include Calico, Cilium, and Flannel.

Overall, network policies are a powerful tool for controlling and securing network traffic in a Kubernetes cluster. By defining rules that allow or deny traffic based on various criteria, network policies help ensure that pods can communicate securely and efficiently, while also providing an additional layer of control and isolation.

# Lab 10: Kubernetes Network Policy

**Topics**

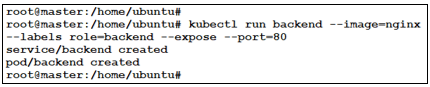
* Network policy with deny all ingress and Pod labels.

## Task 1: Network policy with Pod labels

1. Create a nginx pod and service with **labels role=backend**.

# kubectl run backend --image nginx -l role=backend

# kubectl expose po backend --port 80

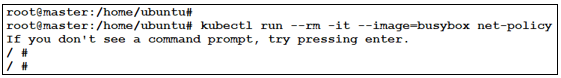


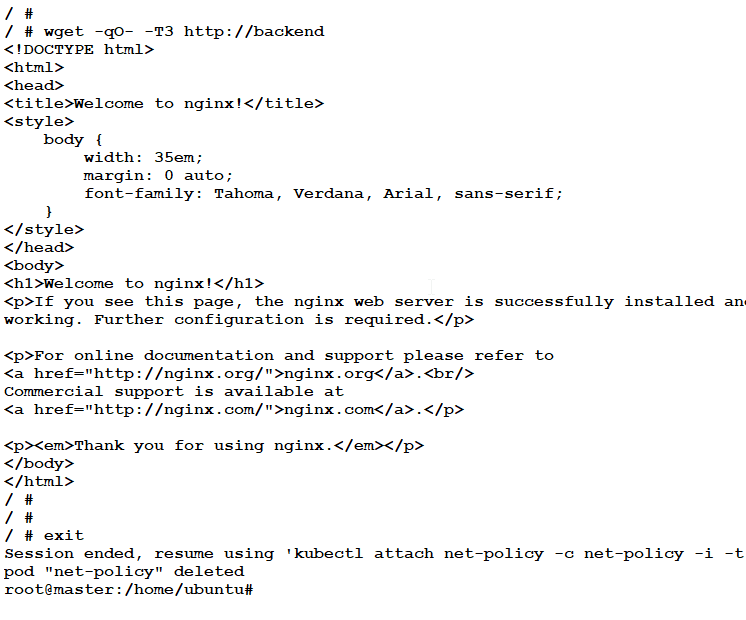
1. Create a new **busybox** pod and verify that it **can** access the backend service.

Then, exit out of the container.

# kubectl run --rm -it --image=busybox net-policy   
# wget -qO- -T3 http://backend

# exit

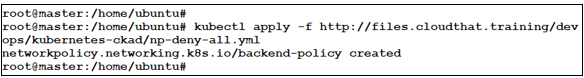




1. Create a network policy which uses labels to deny all ingress traffic.

# kubectl apply -f

http://s3.ap-south-1.amazonaws.com/files.cloudthat.training/devops/kubernetes-ckad/np-deny-all.yml

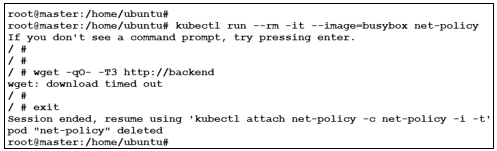


1. Create a new **busybox** pod again and verify that it **cannot** access the backend service.

# kubectl run --rm -it --image=busybox net-policy

# wget -qO- -T3 http://backend

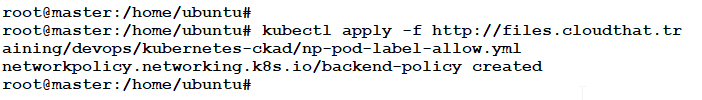
# exit



1. Modify the network policy to allow traffic with matching Pod labels.

Inspect the network policy and notice the selectors.

# kubectl apply -f http://s3.ap-south-1.amazonaws.com/files.cloudthat.training/devops/kubernetes-ckad/np-pod-label-allow.yml



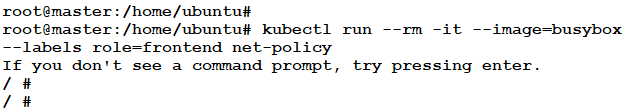
1. Run the **busybox** pod again and verify that it is able to access backend service.

Notice the **labels** on the Pod. Inspect the network policy and notice the selectors.

# kubectl run --rm -it --image=busybox --labels role=frontend net-policy

# wget -qO- -T3 http://backend

# exit



A picture containing bird

Description automatically generated

A picture containing bird

Description automatically generated

1. Run the **busybox** pod again **without** labels and notice that the backend service is not accessible any more.

# kubectl run --rm -it --image=busybox net-policy   
# wget -qO- -T3 http://backend

# exit

A picture containing bird

Description automatically generated

## Task 2: Using WordPress and MySQL as Deployments

1. Create networkdemo namespace.

$ kubectl create ns networkdemo

Text

Description automatically generated with medium confidence

1. Apply the YAML to create mysql first and then wordpress.

$ kubectl create -f http://s3.ap-south-1.amazonaws.com/[files.cloudthat.training/devops/kubernetes-cka/mysql\_deploy.yaml](https://s3.ap-south-1.amazonaws.com/files.cloudthat.training/devops/kubernetes-cka/mysql_deploy.yaml) -n networkdemo

A picture containing graphical user interface

Description automatically generated

$ kubectl create -f http://s3.ap-south-1.amazonaws.com/[files.cloudthat.training/devops/kubernetes-cka/wordpress\_deploy.yaml](https://s3.ap-south-1.amazonaws.com/files.cloudthat.training/devops/kubernetes-cka/wordpress_deploy.yaml) -n networkdemo

A picture containing text

Description automatically generated

Wait for the pods to be in running state and take the NodePort of wordpress service and any of the public IP of the VMs to setup wordpress.

1. Verify that the resources are up and running.

$ kubectl get pod,svc -n networkdemo

Graphical user interface, text

Description automatically generated

1. Open the browser and paste the public IP of the VM followed by the Port in your browser and then follow the steps shown on the browser.

**Example: <PUBLIC IP>:<PORT>**

Graphical user interface, text, application

Description automatically generated

1. Switch back to terminal and create a Deny All ingress networkpolicy and try to access application.

NOTE: you should not be able to access wordpress

$ vim deny-all.yaml

Text

Description automatically generated with medium confidence

apiVersion: networking.k8s.io/v1

kind: NetworkPolicy

metadata:

name: default-deny-ingress

namespace: networkdemo

spec:

podSelector:

matchLabels: {}

policyTypes:

- Ingress

Text

Description automatically generated

1. Apply deny-all.yaml to create an ingress policy.

$ kubectl apply -f deny-all.yaml -n networkdemo

Diagram

Description automatically generated with low confidence

1. Switch to the browser and refresh.

Graphical user interface, application

Description automatically generated

# Nitty Gritty of Role Based Access Control (RBAC) in Kubernetes

Role-Based Access Control (RBAC) is a Kubernetes feature that allows you to control access to the Kubernetes API by defining roles and permissions. It allows you to define different access levels for different users or groups and ensures that only authorized users can perform specific actions in the Kubernetes cluster.

Here's how RBAC works in Kubernetes:

1. **Define Roles:** You can create one or more roles that define a set of permissions that a user or group can have in the cluster. These roles can be cluster-wide or namespace-specific. You can define roles using Kubernetes API objects such as Role, ClusterRole, RoleBinding, and ClusterRoleBinding.
2. **Assign Roles:** You can then assign these roles to users or groups using RoleBindings or ClusterRoleBindings. RoleBindings allow you to assign roles to users or groups within a namespace, while ClusterRoleBindings allow you to assign roles to users or groups across the entire cluster.
3. **Role Permissions:** Each role can specify a set of permissions that define what a user or group is allowed to do in the cluster. For example, you can create a role that allows users to read pods, but not modify or delete them. Permissions are defined using Kubernetes API objects such as APIGroups, Resources, Verbs, and ResourceNames.
4. **Access Control:** Once the roles and permissions are defined, Kubernetes enforces access control by checking the user's credentials and the permissions associated with their assigned roles when they attempt to access the Kubernetes API. If the user has the necessary permissions, they are allowed to perform the action. Otherwise, the action is denied.

In a nutshell RBAC is a powerful feature of Kubernetes that allows you to define fine-grained access control for your cluster. By defining roles and permissions, you can control who can access your cluster and what actions they can perform. This helps to improve security and reduce the risk of unauthorized access or data breaches.

Top of Form

# Lab 11: Control access using RBAC

Provide necessary permissions to pod test-01 inside purple namespace so that it is able perform a GET request to Kubernetes API server.

## Task 1: Create a new ServiceAccount demo-sa

1. See the available namespaces.

$ kubectl get ns

Text, letter

Description automatically generated

1. Create a new namespace purple and a ServiceAccount for that namespace.

$ kubectl create ns purple

Text

Description automatically generated with low confidence

$ kubectl create sa -n purple demo-sa

Text

Description automatically generated

1. Verify that the namespace and service account has been created.

$ kubectl get ns

Text

Description automatically generated

Text, table

Description automatically generated

## Task 2: Create a new Role and RoleBinding

1. Create new Role and RoleBinding.

$ kubectl create role demo-role --verb=list --resource=pods -n purple

Text

Description automatically generated

$ kubectl create rolebinding demo-rb --role=demo-role --serviceaccount=purple:demo-sa -n purple

**Text

Description automatically generated with medium confidence**

## Task 3: Test whether you can use a GET request to Kubernetes API

1. Create a pod with default serviceaccount in purple namespace.

$ kubectl run test --image=nginx -n purple

A picture containing diagram

Description automatically generated

1. exec into the pod created in previous step.

$ kubectl exec -it -n purple test -- /bin/bash



1. CURL to the Kube-api server to see whether you are able to list the pods running in purple namespace.

# curl -v --cacert /var/run/secrets/kubernetes.io/serviceaccount/ca.crt -H "Authorization: Bearer $(cat /var/run/secrets/kubernetes.io/serviceaccount/token)" https://kubernetes.default/api/v1/namespaces/purple/pods

Text

Description automatically generated

Graphical user interface, text, application, email

Description automatically generated

1. Exit from the pod.

Diagram

Description automatically generated with low confidence

1. Now create a pod with demo-sa serviceaccount that has the escalated privileges.

$ vi pod-sa.yaml

apiVersion: v1

kind: Pod

metadata:

name: test-01

namespace: purple

spec:

serviceAccountName: demo-sa

containers:

- name: my-container

image: nginx

$ kubectl apply -f pod-sa.yaml

1. Exec into the pod.

$ kubectl exec -it -n purple test-01 -- /bin/bash

A picture containing diagram

Description automatically generated

1. Now CURL to kube-api server and see whether you can list pods.

# curl -v --cacert /var/run/secrets/kubernetes.io/serviceaccount/ca.crt -H "Authorization: Bearer $(cat /var/run/secrets/kubernetes.io/serviceaccount/token)" https://kubernetes.default/api/v1/namespaces/purple/pods

A screenshot of a computer

Description automatically generated with medium confidence

Text, letter

Description automatically generated

1. Exit from the pod.

**A picture containing schematic

Description automatically generated**

You will see that it will list the pods in the purple namespace that we were earlier not able to list.

# Concept of Pod Scheduling

Pod scheduling in Kubernetes is the process of determining which nodes in the cluster should run a particular pod. When a pod is created, the Kubernetes scheduler is responsible for finding a suitable node to run it based on various factors such as resource requirements, affinity and anti-affinity rules, node selectors, and taints and tolerations.

The pod scheduling process involves the following steps:

1. Resource requirements: Kubernetes scheduler considers the resource requirements of the pod, such as CPU and memory requests and limits, and looks for a node that has enough available resources to run the pod. If no such node is available, the pod remains in the Pending state until resources become available.
2. Affinity and anti-affinity: Affinity rules specify which nodes the pod should be scheduled on based on certain attributes, such as the node’s labels or the presence of other pods on the same node. Anti-affinity rules, on the other hand, specify which nodes the pod should not be scheduled on based on similar attributes.
3. Node selectors: Node selectors allow you to specify which nodes the pod should be scheduled on based on certain labels assigned to the nodes. If the node does not have the required label, the pod will not be scheduled on that node.
4. Taints and tolerations: Taints are used to repel pods from a node, while tolerations are used to attract pods to a node. When a node has a taint, it will not schedule any pods unless the pod has a matching toleration.
5. Priority and preemption: If multiple pods are competing for the same resources, Kubernetes scheduler assigns each pod a priority based on various factors such as resource requirements and QoS class. The pod with the highest priority is scheduled first, while lower priority pods are preempted if necessary to make room for higher priority pods.

In addition to the above factors, Kubernetes scheduler also considers other constraints such as node capacity, node health, and inter-pod communication when scheduling pods.

In summary, pod scheduling in Kubernetes is a complex process that considers various factors such as resource requirements, affinity and anti-affinity rules, node selectors, taints and tolerations, and priority and preemption to ensure efficient and optimal use of cluster resources. By understanding how the scheduler works, you can optimize your Kubernetes applications for better performance and reliability.

# Lab 12: 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

A picture containing graphical user interface

Description automatically generated

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

$ kubectl get nodes --show-labels

A screenshot of a computer

Description automatically generated with medium confidence

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.

Text

Description automatically generated

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

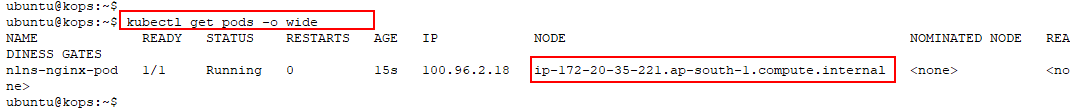
$ kubectl apply -f nlns-pod.yaml

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-

**A picture containing text

Description automatically generated**

1. Verify that the label is deleted.

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



# Lab 13: 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.

Text

Description automatically generated

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

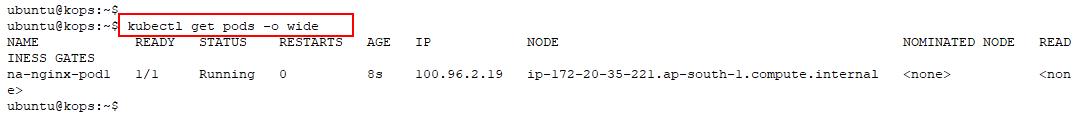
$ kubectl apply -f aff-na-pod1.yaml

A picture containing Word

Description automatically generated

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

A picture containing graphical user interface

Description automatically generated

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.

Graphical user interface, text

Description automatically generated

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

Logo

Description automatically generated with medium confidence

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

$ kubectl get nodes

Text, letter

Description automatically generated

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

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

A picture containing graphical user interface

Description automatically generated

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

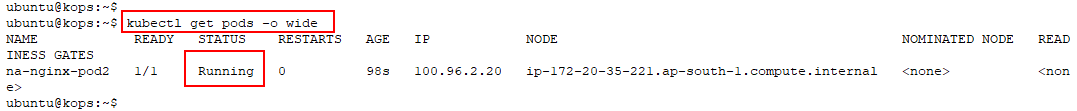
$ kubectl apply -f aff-na-pod2.yaml

A picture containing logo

Description automatically generated

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

A picture containing graphical user interface

Description automatically generated

## 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.

Text, letter

Description automatically generated

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

Graphical user interface

Description automatically generated with medium confidence

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.

Text

Description automatically generated

1. Apply the yaml created in the previous step.

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

Logo

Description automatically generated with medium confidence

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

Graphical user interface

Description automatically generated with medium confidence

## 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

Graphical user interface, text

Description automatically generated with medium confidence

# Introduction to Helm

Helm is a package manager for Kubernetes that makes it easy to install, manage, and upgrade applications on a Kubernetes cluster. Helm uses charts, which are packages that contain all the necessary resources and configuration files to deploy and manage a Kubernetes application.

Here are some key concepts and features of Helm and Helm charts:

* Package manager: Helm is a package manager for Kubernetes, like apt or yum for Linux. It allows you to easily install and manage Kubernetes applications, without having to manually create and manage all the necessary resources and configuration files.
* Charts: A Helm chart is a package that contains all the necessary resources and configuration files to deploy and manage a Kubernetes application. A chart can contain multiple files, including Kubernetes manifests, configuration files, and templates.
* Versioning: Helm charts are versioned, allowing you to track changes and upgrades to your application over time. You can use Helm to install or upgrade specific versions of a chart, or to rollback to a previous version if needed.
* Templates: Helm charts use templates to generate Kubernetes manifests and configuration files. Templates allow you to define variables and parameters that can be customized at installation time, making it easy to deploy the same chart with different configurations.
* Repositories: Helm uses repositories to store and distribute charts. You can create your own private repository or use one of the public repositories available, such as the official Helm repository or the Bitnami repository.
* Helm CLI: Helm provides a command-line interface (CLI) that allows you to install, upgrade, and manage Helm charts on a Kubernetes cluster. The Helm CLI includes commands for managing repositories, searching for charts, installing and upgrading charts, and rolling back to previous versions.

Overall, Helm is a powerful tool for managing Kubernetes applications, and Helm charts provide a standardized way to package and distribute Kubernetes applications. By using Helm and Helm charts, you can simplify the deployment and management of your Kubernetes applications, and easily manage upgrades and rollbacks over time.

# Lab 14: Installing WordPress with Helm

This lab demonstrates setting up WordPress on Kubernetes using Helm.

## Task 1: Helm setup

1. Download the shell script for the installation of the Helm package manager and run it.

$ wget files.cloudthat.training/devops/kubernetes-essentials/helm.sh

$ bash helm.sh

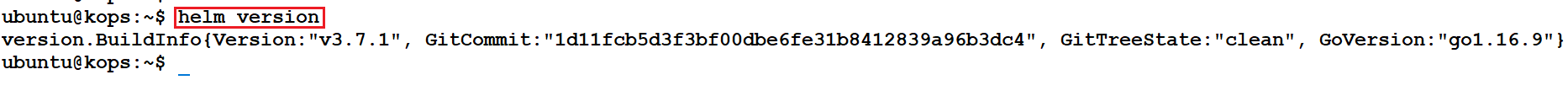
$ helm version

A picture containing diagram

Description automatically generated

Text

Description automatically generated



## Task 2: Setup WordPress

1. Add bitnami repository to helm which contains WordPress chart.

$ helm repo add bitnami https://charts.bitnami.com/bitnami

$ helm repo update

$ helm repo list

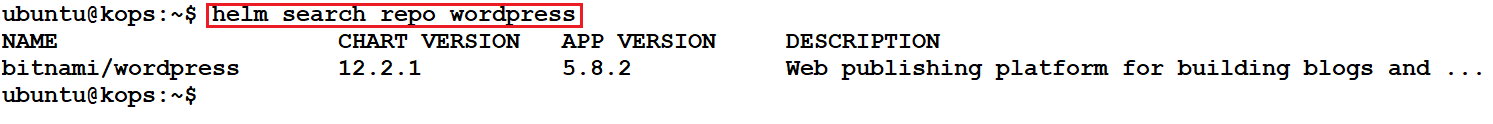
Graphical user interface, text, email

Description automatically generated

1. Search for WordPress package from the repository and install the WordPress chart with release name wordpress-chart. You can name it as you want.

$ helm search repo wordpress

$ helm install wordpress-chart bitnami/wordpress



Text

Description automatically generated

1. Perform helm fetch to get the WordPress compressed file to the working directory. Perform ls and verify that a compressed WordPress file is present in PWD.

$ helm fetch bitnami/wordpress

$ ls

Text

Description automatically generated with low confidence

1. Extract the compressed file using the below command.

$ tar -xvzf wordpress-12.2.1.tgz

Text

Description automatically generated

A close-up of a document

Description automatically generated with low confidence

1. Now, deploy the WordPress using helm install command. This will set up the pods and services for WordPress.

$ helm install wordpress --generate-name

Text

Description automatically generated

## Task 3: Verify that WordPress has been set up

1. Verify that the pods are running.

$ kubectl get pods

Table

Description automatically generated with medium confidence

1. Now Notice that MariaDB (database) pods are part of statefulset.

$ kubectl get sts

$ kubectl describe sts

Text

Description automatically generated with medium confidence

Text

Description automatically generated

Graphical user interface, text, application

Description automatically generated

1. Also Notice that the front end of wordpress are part of deployment.

$ kubectl get deploy

Text

Description automatically generated

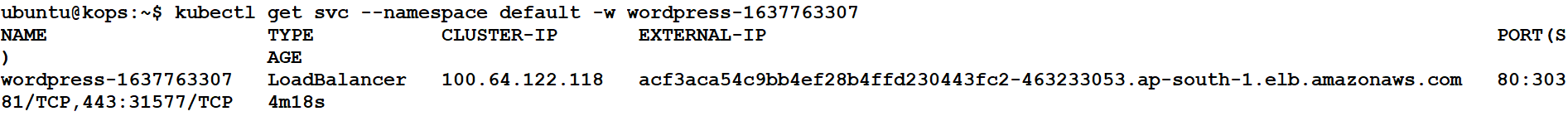
1. View the services using the below commands.
2. Make note of the EXTERNAL IP of the LoadBalancer service.

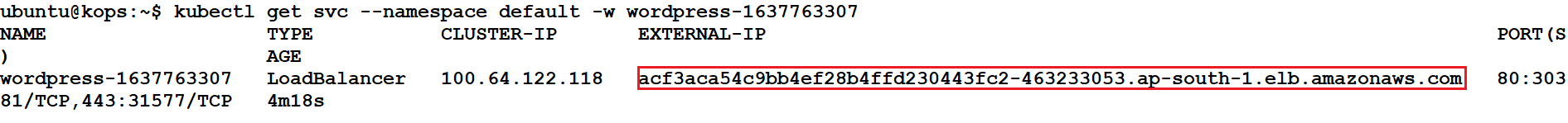
$ kubectl get svc

$ kubectl get svc --namespace default -w wordpress-1637763307

Table

Description automatically generated with medium confidence





1. Open the browser and paste the service endpoint noted on the previous step. Observe that the WordPress site is up and running.

Graphical user interface, text, application, email

Description automatically generated

## Task 4: Cleanup

1. List the current helm release and delete it

$ helm ls

$ helm delete <helm\_Release\_Name >

$ helm ls

**A picture containing text

Description automatically generated**

Graphical user interface, text, application, email

Description automatically generated