**Kubernetes in GCP (Google Cloud Platform)**

Table of Contents

1 Introduction to Kubernetes3

1.1 What is kubernetes?3

1.2 Why kubernetes?3

1.3 Kubernetes Architecture4

1.4 Things to know before starting with kubernetes5

1.5 Some commonly used kubectl commands7

2 Setup kubernetes in GCP VM8

2.1 Creating kubernetes cluster 8

2.2 Installing Kubernetes 11

2.3 Connecting with cluster12

3 Deploying selenium hub in kubernetes cluster13

3.1 Creating YAML files for deployment14

3.2 Deploy YAML files15

3.2.1 Through commands15

3.2.2. Through Jenkins job17

4 Deploying zalenium in kubernetes cluster19

4.1 Creating YAML files for deployment19

4.2 Deploy YAML files21

4.2.1 Through commands21

5 Jenkins job creation for docker image building25

6 Creating dockerfile with arguments27

7 Jenkins job creation for docker image building with parameters28

1. Introduction to Kubernetes

1.1 What is Kubernetes?

Kubernetes is a portable, extensible open-source platform for managing containerized workloads and services, that facilitates both declarative configuration and automation. It has a large, rapidly growing ecosystem. Kubernetes services, support, and tools are widely available.

Google open-sourced the Kubernetes project in 2014.

Kubernetes provides a container-centric management environment. It orchestrates computing, networking, and storage infrastructure on behalf of user workloads. This provides much of the simplicity of Platform as a Service (PaaS) with the flexibility of Infrastructure as a Service (IaaS), and enables portability across infrastructure providers.

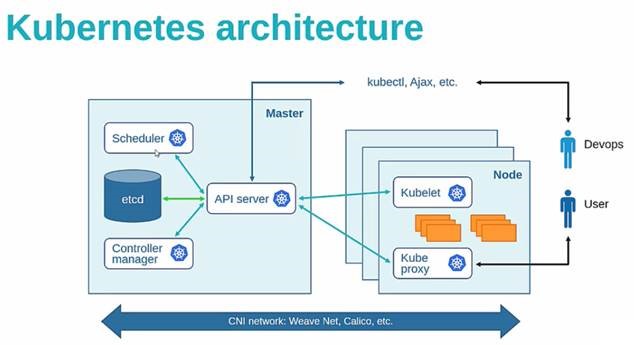
Kubernetes was also designed to serve as a platform for building an ecosystem of components and tools to make it easier to deploy, scale, and manage applications.

1.2 Why Kubernetes?

It helps with things like:

* Running containers across many different machines.
* Scaling up or down by adding or removing containers when demand changes.
* Keeping storage consistent with multiple instances of an application.
* Distributing load between the containers.
* Launching new containers on different machines if something fails.

1.3 Kubernetes Architecture



Here,

**Nodes:**  Nodes might be physical machines or VMs. Again, the idea is abstraction: whatever the app is running on, Kubernetes handles deployment on that substrate. It is also possible to ensure that certain containers run only on VMs or only on bare metal. Nodes run pods.

**Master:** The Kubernetes Master controls and coordinates all the nodes in the cluster with the help of some processes that run on one or more master nodes in the cluster.

**Scheduler:** The scheduler parcels out workloads to nodes so that they’re balanced across resources and so that deployments meet the requirements of the application definitions.

**Controller manager:** The controller manager ensures the state of the system—applications, workloads, etc.—matches the desired state defined in Etcd’s configuration settings.

**Etcd:** It is an open source key-value store which stores configuration data accessed by all nodes (node and master) in the cluster such as cluster state. It communicates with the most component to receive commands and work. It also manages network rules and port forwarding activity.

**API Server:**  It generally validates the configuration data store in ‘Etcd’ and the details of the deployed container that are in agreement. The API server acts as an entry point for all the REST commands used for controlling the cluster.

**Kube-Proxy**: It is the network proxy which runs on each worker node and listens to the API server for each Service endpoint creation/deletion. For each Service endpoint, kube-proxy sets up the routes so that it can reach to it.

**Kubelet:** It is an agent which communicates with the Master node and executes on nodes or the worker nodes. It gets the Pod specifications through the API server and executes the containers associated with the Pod and ensures that the containers described in those Pod are running and healthy.

1.4 Things to know before starting with Kubernetes.

**Cluster:**  Cluster refers to the group of machines running Kubernetes (itself a clustered application) and the containers managed by it. A Kubernetes cluster must have a master, the system that commands and controls all the other Kubernetes machines in the cluster. A highly available Kubernetes cluster replicates the master’s facilities across multiple machines. But only one master at a time runs the job scheduler and controller-manager

Each cluster contains Kubernetes nodes.

**Pods:** pods are the most basic Kubernetes objects that can be created or managed. Each pod represents a single instance of an application or running process in Kubernetes, and consists of one or more containers. Kubernetes starts, stops, and replicates all containers in a pod as a group. Pods keep the user’s attention on the application, rather than on the containers themselves. Details about how Kubernetes needs to be configured, from the state of pods on up, is kept in Etcd, a distributed key-value store.

Pods are created and destroyed on nodes as needed to conform to the desired state specified by the user in the pod definition.

**Controller:** Kubernetes provides an abstraction called a controller for dealing with the logistics of how pods are spun up, rolled out, and spun down. Controllers come in a few different flavors depending on the kind of application being managed. For instance, the recently introduced “StatefulSet” controller is used to deal with applications that need persistent state. Another kind of controller, the deployment, is used to scale an app up or down, update an app to a new version, or roll back an app to a known-good version if there’s a problem.

**Namespaces:** They are virtual clusters backed by the physical cluster.

**Volume:** It is essentially a directory accessible to all containers running in a Pod.

**PersistentVolume (PV):** It is a piece of storage in the cluster that has been provisioned by an administrator or dynamically provisioned using Storage Classes. It is a resource in the cluster. PVs are volume plugins like Volumes, but have a lifecycle independent of any individual Pod that uses the PV. This API object captures the details of the implementation of the storage, be that NFS, iSCSI, or a cloud-provider-specific storage system.

**PersistentVolumeClaim (PVC):** It is a request for storage by a user. It is similar to a Pod. Pods consume node resources and PVCs consume PV resources. Pods can request specific levels of resources (CPU and Memory). Claims can request specific size and access modes (e.g., they can be mounted once read/write or many times read-only).

**ServiceAccount:** It provides an identity for processes that run in a Pod. Every namespace has a default service account resource called default. When someone access the cluster (for example, using kubectl), you are authenticated by the apiserver as a particular User Account. Processes in containers inside pods can also contact the apiserver. When they do, they are authenticated as a particular Service Account (for example, default).

**Role and ClusterRole:** A ServiceAccount is not that useful unless certain rights are bound to it. Rights are known as Role or ClusterRole in Kubernetes. A Role (the same applies to a ClusterRole) contains a list of rules. Each rule defines some actions that can be performed (e.g: list, get, watch) against a list of resources (e.g: Pod, Service, Secret) within apiGroups (eg: core, apps/v1). While a Role defines rights for a specific namespace, the scope of a ClusterRole is the entire cluster.

**RoleBinding and ClusterRoleBinding:** A role binding grants the permissions defined in a role to a user or set of users. It holds a list of subjects (users, groups, or service accounts), and a reference to the role being granted. Permissions can be granted within a namespace with a RoleBinding, or cluster-wide with a ClusterRoleBinding. ClusterRoleBinding may be used to grant permission at the cluster level and in all namespaces

A RoleBinding may reference a Role in the same namespace. For example, a RoleBinding which grants the “pod-reader” role to the user “jane” within the “default” namespace allows “jane” to read pods in the “default” namespace.

**Service:** Kubernetes provides an abstraction called a service for dealing with the application lifecycle. A service describes how a given group of pods (or other Kubernetes objects) can be accessed via the network. The pods that constitute the back end of an application might change, but the front end shouldn’t have to know about that or track it. Services make this possible. Services run continuously until it is stopped explicitly. Kubernetes allows you to specify what kind of service you want. The default is ClusterIP. Other types are NodePort and LoadBalancer.

**ClusterIP:** Exposes the service on a cluster-internal IP. Choosing this value makes the service only reachable from within the cluster.

**NodePort:** Exposes the service on each Node’s IP at a static port (the NodePort). A ClusterIP service, to which the NodePort service will route, is automatically created. You’ll be able to contact the NodePort service, from outside the cluster, by requesting <NodeIP>:<NodePort>.

**LoadBalancer:** Exposes the service externally using a cloud provider’s load balancer. NodePort and ClusterIP services, to which the external load balancer will route, are automatically created.

**Kubectl: -** It controls the Kubernetes Cluster. It is one of the key components of Kubernetes which runs on the workstation on any machine when the setup is done. It has the capability to manage the nodes in the cluster.

1.3 Some commonly used kubectl commands

Kubectl commands are used to interact and manage Kubernetes objects and the cluster. Following are some commands used in Kubernetes via kubectl.

* kubectl apply − It has the capability to configure a resource by file or stdin.

$ kubectl apply –f <filename>

* kubectl attach − This attaches things to the running container.

$ kubectl attach <pod> –c <container>

$ kubectl attach 123456-7890 -c tomcat-conatiner

* kubectl cluster-info − It displays the cluster Info.

$ kubectl cluster-info

* kubectl create − To create resource by filename of or stdin. To do this, JSON or YAML formats are accepted.

$ kubectl create –f <File Name>

$ cat <file name> | kubectl create –f -

* kubectl delete − Deletes resources by file name, stdin, resource and names.

$ kubectl delete –f ([-f FILENAME] | TYPE [(NAME | -l label | --all)])

* kubectl describe − Describes any particular resource in kubernetes. Shows details of resource or a group of resources.

$ kubectl describe <type> <type name>

$ kubectl describe pod tomcat

* kubectl exec − This helps to execute a command in the container.

$ kubectl exec POD <-c CONTAINER > -- COMMAND < args...>

$ kubectl exec tomcat 123-5-456 date

* kubectl expose − This is used to expose the Kubernetes objects such as pod, replication controller, and service as a new Kubernetes service. This has the capability to expose it via a running container or from a yaml file.

$ kubectl expose (-f FILENAME | TYPE NAME) [--port=port] [--protocol = TCP|UDP]

[--target-port = number-or-name] [--name = name] [--external-ip = external-ip-ofservice]

[--type = type]

$ kubectl expose rc tomcat –-port=80 –target-port = 30000

$ kubectl expose –f tomcat.yaml –port = 80 –target-port =

* kubectl get − This command is capable of fetching data on the cluster about the Kubernetes resources.

$ kubectl get [(-o|--output=)json|yaml|wide|custom-columns=...|custom-columnsfile=...|

go-template=...|go-template-file=...|jsonpath=...|jsonpath-file=...]

(TYPE [NAME | -l label] | TYPE/NAME ...) [flags]

For example,

$ kubectl get pod <pod name>

$ kubectl get service <Service name>

* kubectl logs − They are used to get the logs of the container in a pod. Printing the logs can be defining the container name in the pod. If the POD has only one container there is no need to define its name.

$ kubectl logs [-f] [-p] POD [-c CONTAINER]

Example

$ kubectl logs tomcat.

$ kubectl logs –p –c tomcat.8

* kubectl scale − It will scale the size of Kubernetes Deployments, ReplicaSet, Replication Controller, or job.

$ kubectl scale [--resource-version = version] [--current-replicas = count] --

replicas = COUNT (-f FILENAME | TYPE NAME )

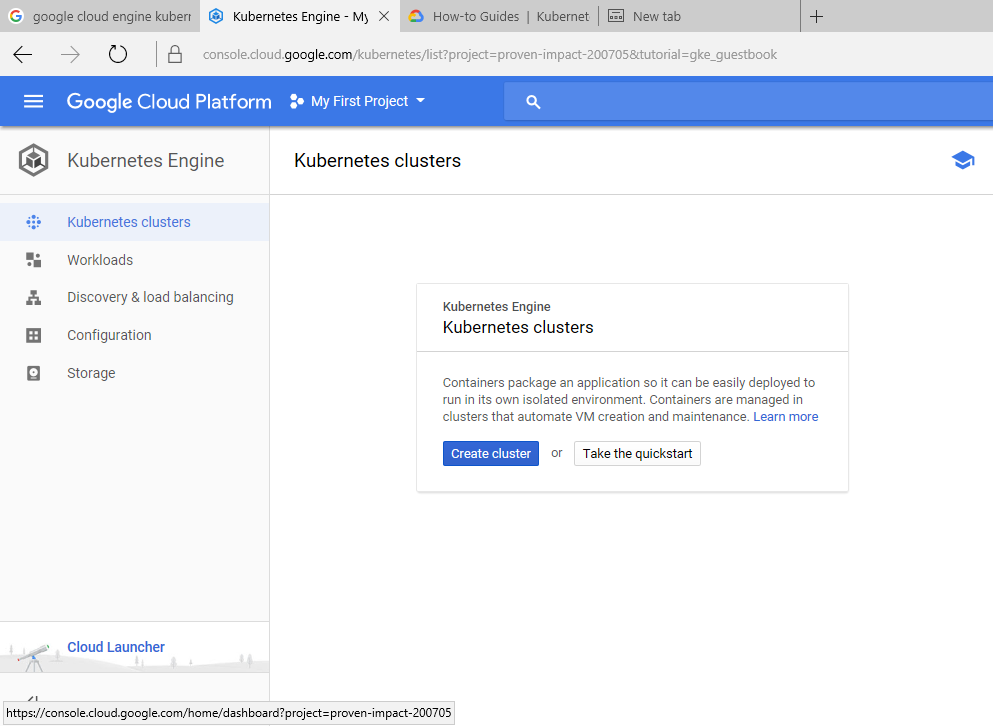
$ kubectl scale –-replica = 3 rs/tomcat

$ kubectl scale –replica = 3 tomcat.yaml

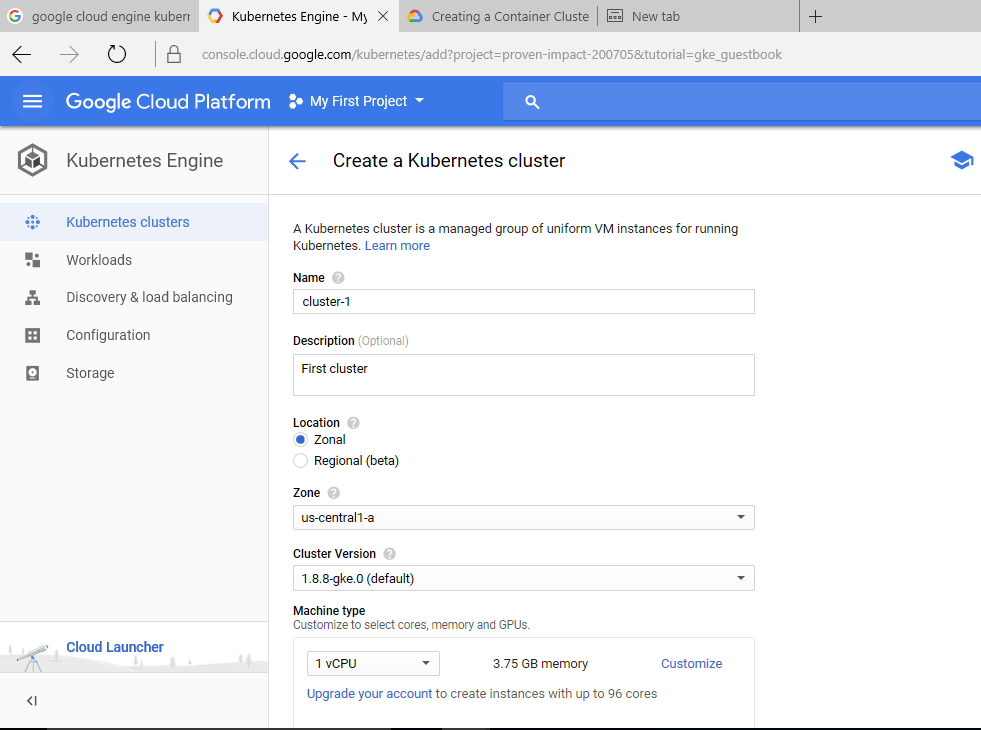
1. Setup kubernetes in GCP VM

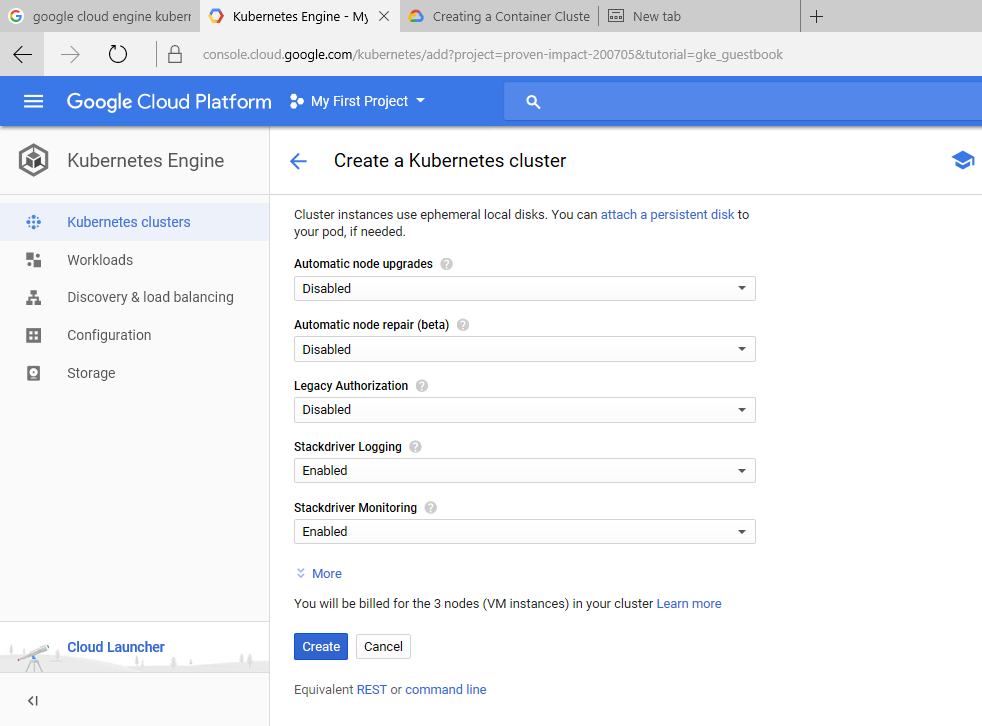
2.1 Creating kubernetes cluster

Click on Kubernete engine->kubernetes clusters and you would see the following screen.

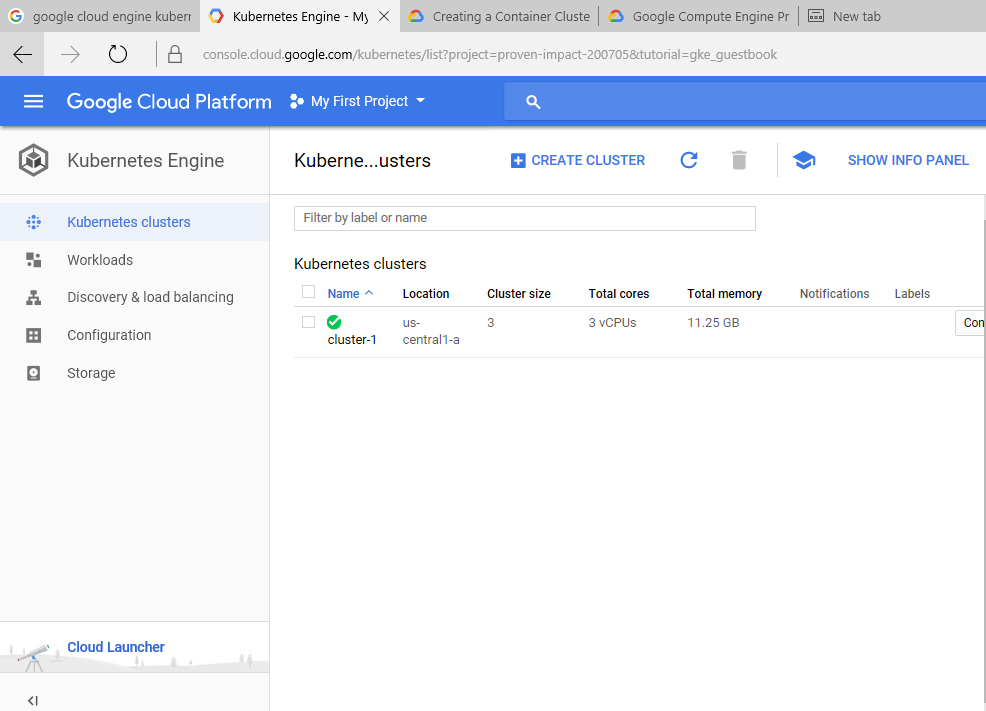


Click on create cluster by providing any name for it as follows: -





After creating, you should be able to see the cluster’s name listed as follows: -

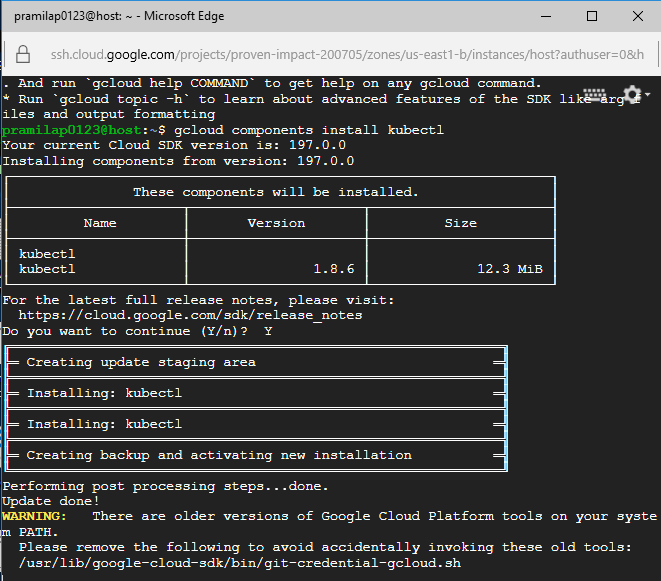


To connect the created cluster with your VM for creating nods and pods in it, you have to connect with gCloud.

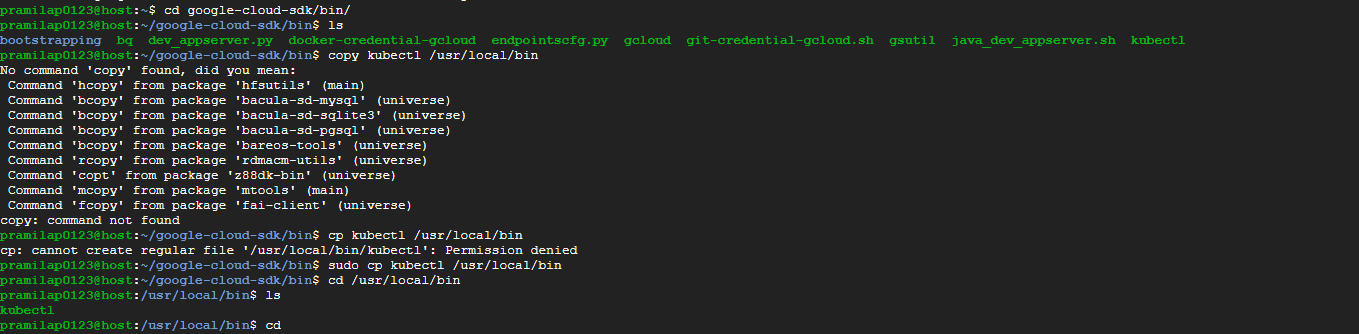
2.2 Installing Kubernetes

You should create VM instance in GCP and install gCloud and configure it using gcloud init command (refer GCE\_Kubernetes document for gcloud init where you have to provide your gmail account name and zone).

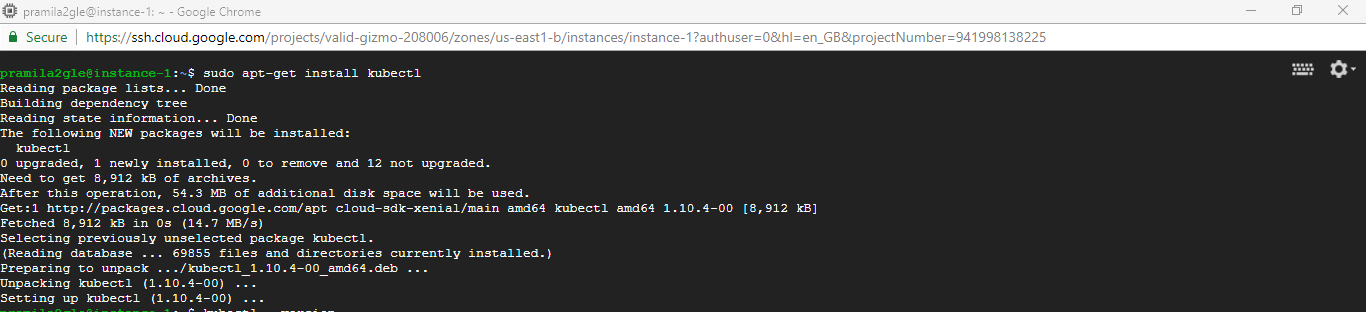
Now install kubernetes as below: -



Kubectl is installed but if you are not able to execute commands then copy the kubectl folder from gcloud bin folder to usr/bin folder as follows: -

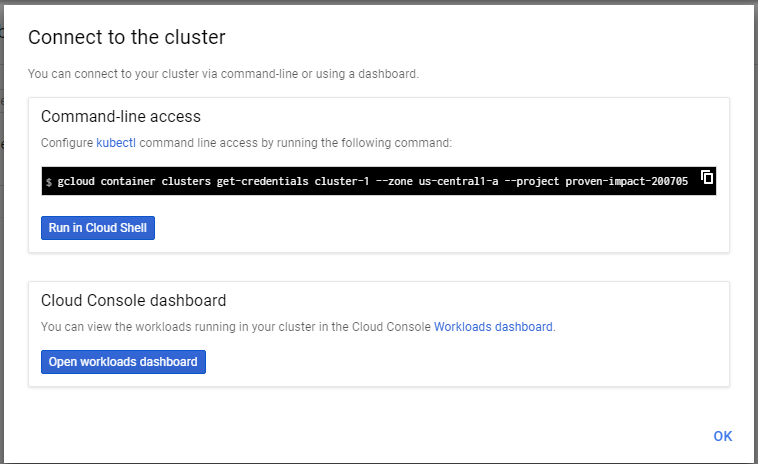


If you are not able to install kubectl through gcloud component command while installing kubectl then try this command

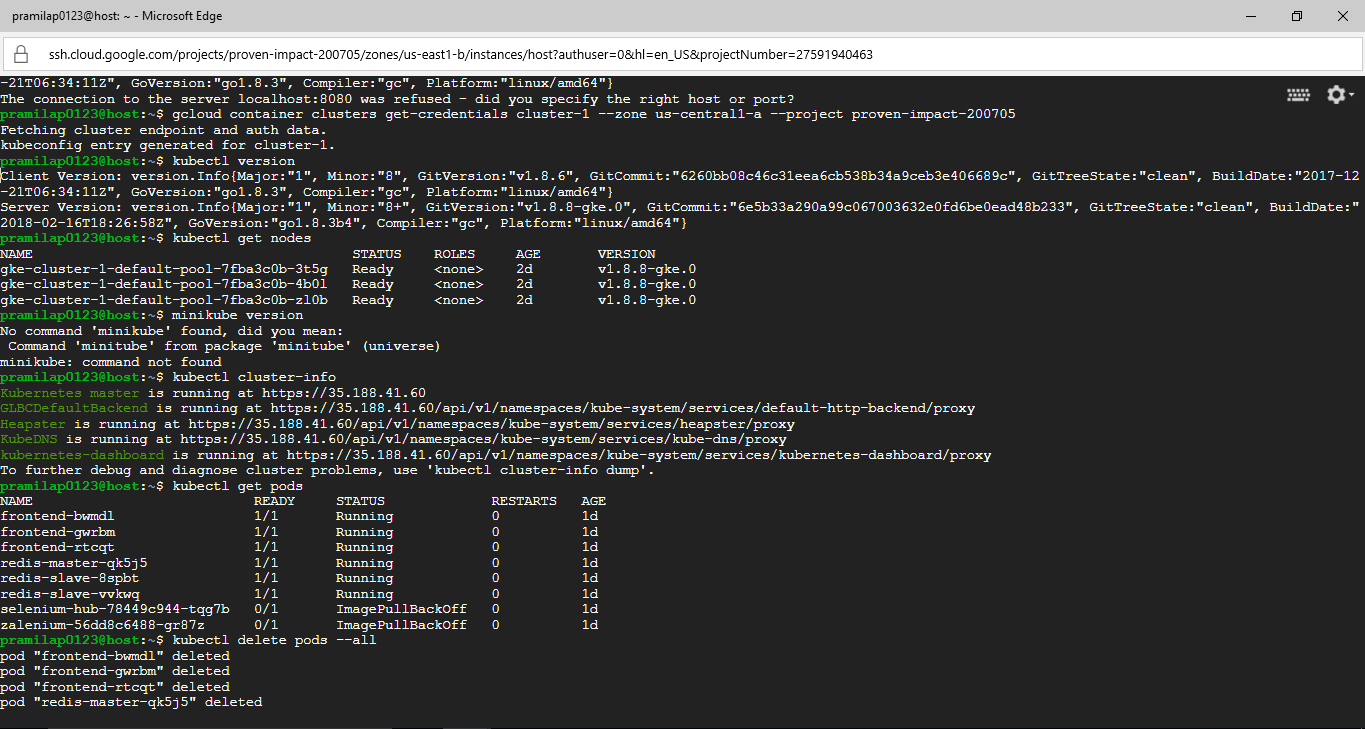


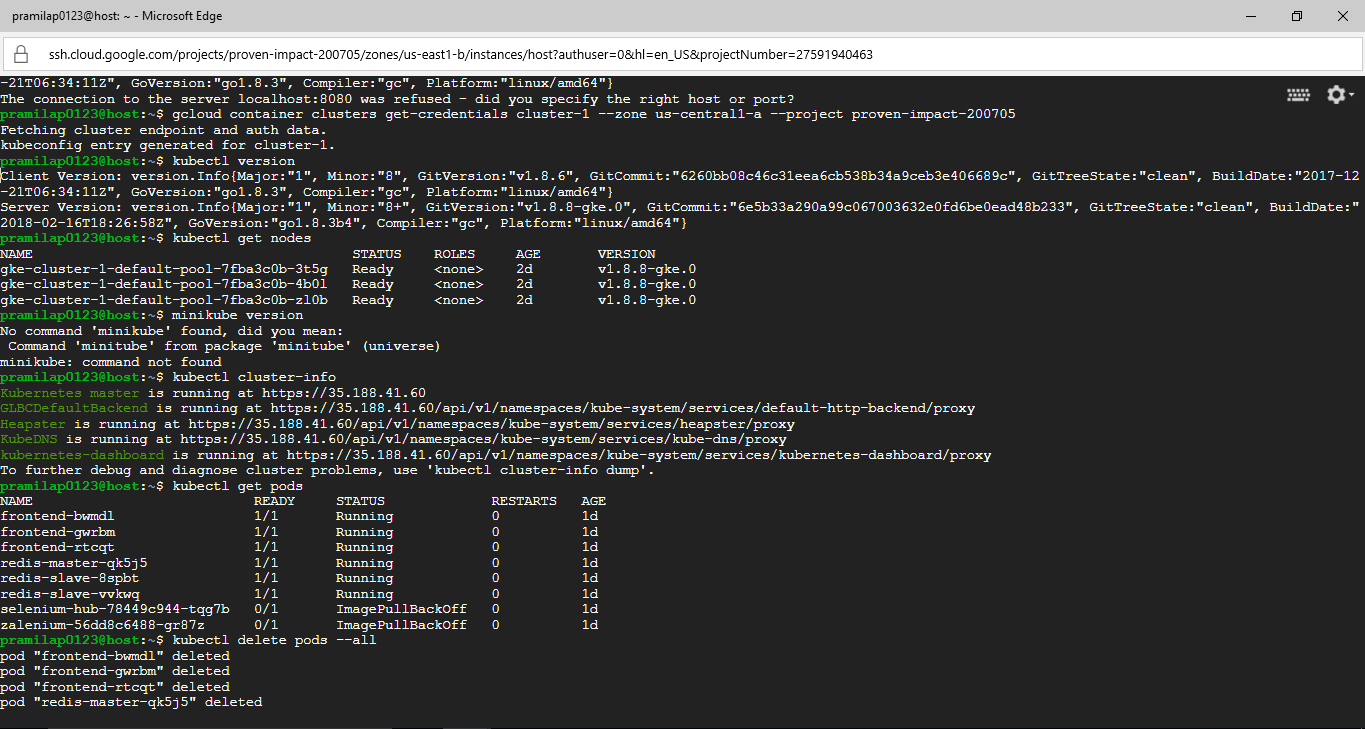
2.3 Connecting with Kubernetes cluster

To connect to the created kubernetes cluster, click on connect button of the corresponding cluster, then a window will show up like this



Copy the line under command-line access and paste it in your VM. Then press enter.



When cluster is configured check whether you can see the nodes created in it as follows: 

So, now we can work with kubernetes cluster through the VM.

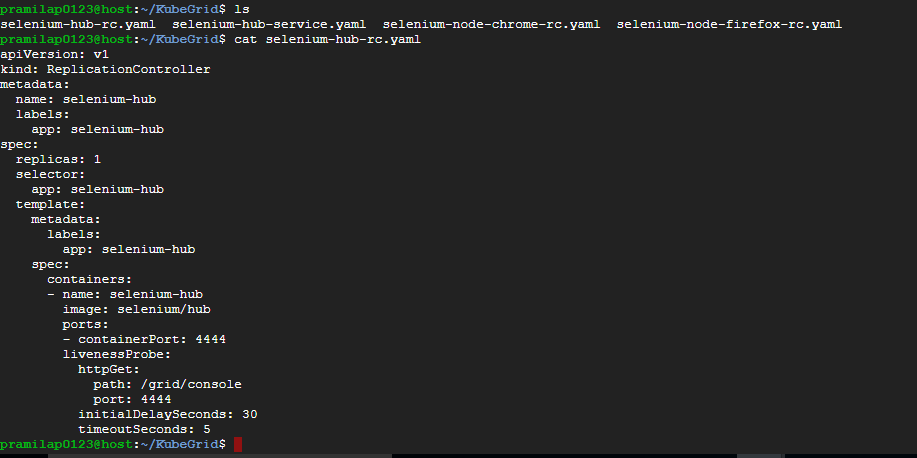
1. Deploying selenium hub in kubernetes cluster

3.1 Creating YAML files for deployment

kubectl, Docker, Jenkins (Which requires open jdk and nginx installed) should be installed in VM for deploying selenium hub in cluster.

To deploy selenium hub in Kubernetes cluster, we have to create YAML files (which defines needed configurations) for selenium hub deployment, its service (to execute testcases), selenium chrome node deployment and firefox node deployment. Create a folder and create 4 YAML files inside it for each.

YAML file for selenium hub deployment is as follows: -



So here, we have specified the api version as v1(but for deployments we specify as apps/v1 or extensions/v1beta1 as they are not included in v1) then we have specified its kind as we want to replicate the nodes.

Next, we’re specifying that we want to create a Replication Controller; we might specify instead a Deployment, Job, Service, and so on, depending on what we’re trying to achieve.

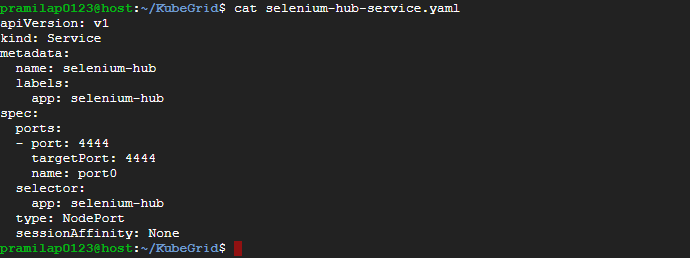
Next we specify the metadata. Here we’re specifying the name of the Replication Controller, as well as the label we’ll use to identify the Replication Controller to Kubernetes.

Finally, we’ll specify the actual objects that make up the pod. The spec property includes any containers, storage volumes, or other pieces that Kubernetes needs to know about, as well as properties such as whether to restart the container if it fails.

So, here, we have specified how many replicas we need, container’s name, image name, port in which it will be hosted, its path to access and timeout seconds.

With this YAML file, selenium-hub will be deployed. But since we need the selenium hub everytime to execute our testcases, we will create its service as well.

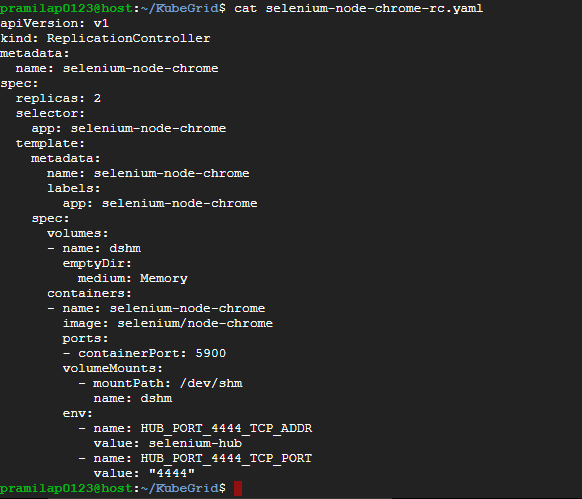
YAML file for selenium hub service:



Here, we have specified the same api version and mentioned that we want to create service.

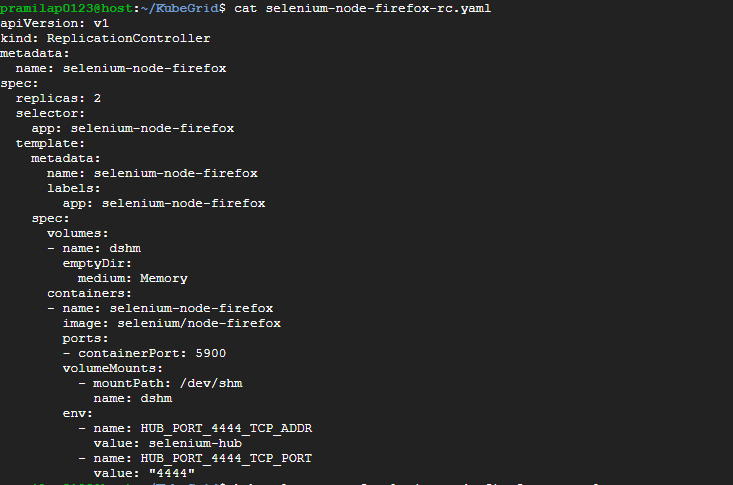
Then, under metadata, we are mentioned its name and label. Next, we have specified the port number and the type of service as NodePort.

YAM file for chrome node deployment:



Here, it is a replication controller and its name is specified. We have given replica’s count as 2 as we want 2 chrome nodes to be created initially. Inside spec we have used volumes attribute to create an empty directory named dshm with medium memory and mounting the container’s memory inside it using VolumeMounts attribute as chrome may want to cache some data while executing testcases for multiple tabs. Env attribute is used to specify that its wants selenium hub to run and connect to it with the mentioned port and name

YAML file for firefox node deployment:

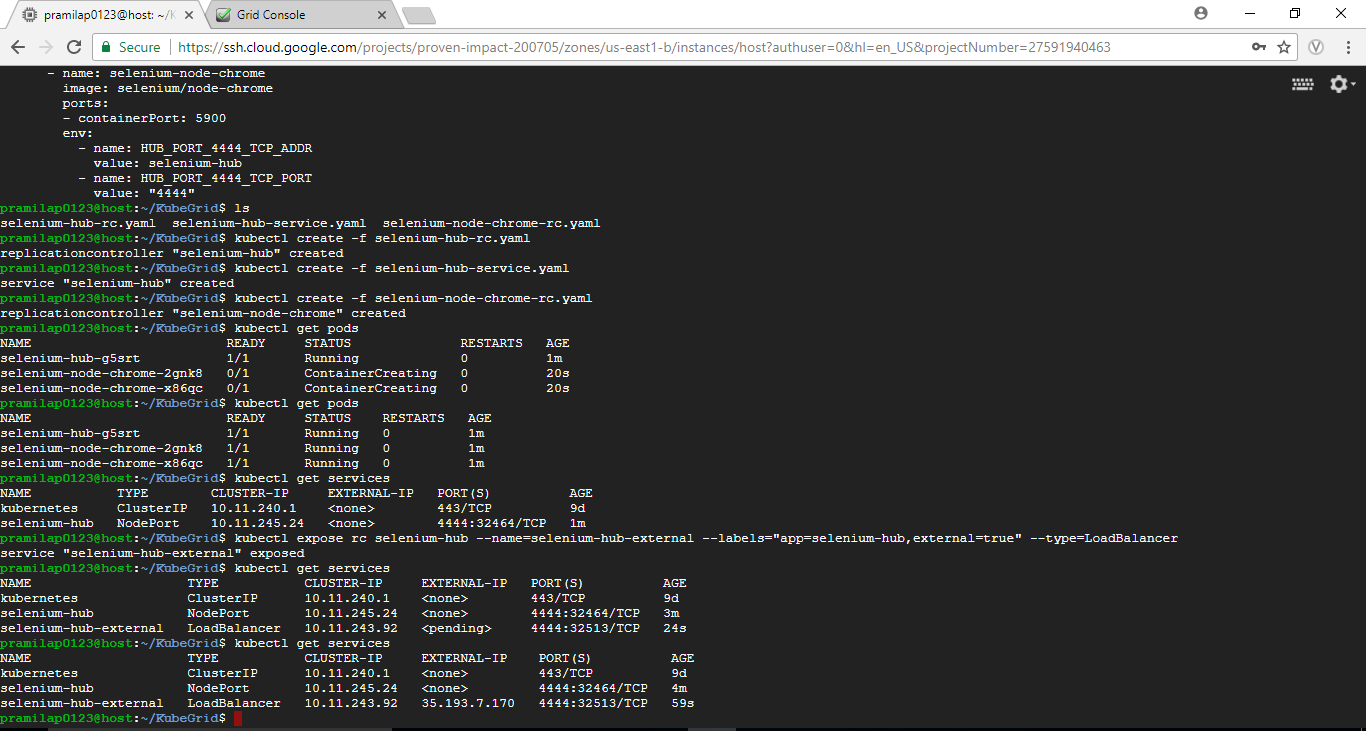


YAML file for firefox is similar to chrome, just the name is different.

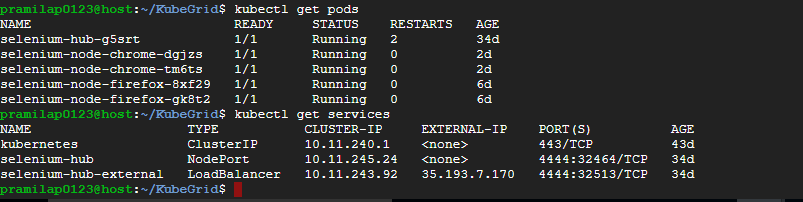
3.2 Deploy YAML files

3.2.1 Through commands

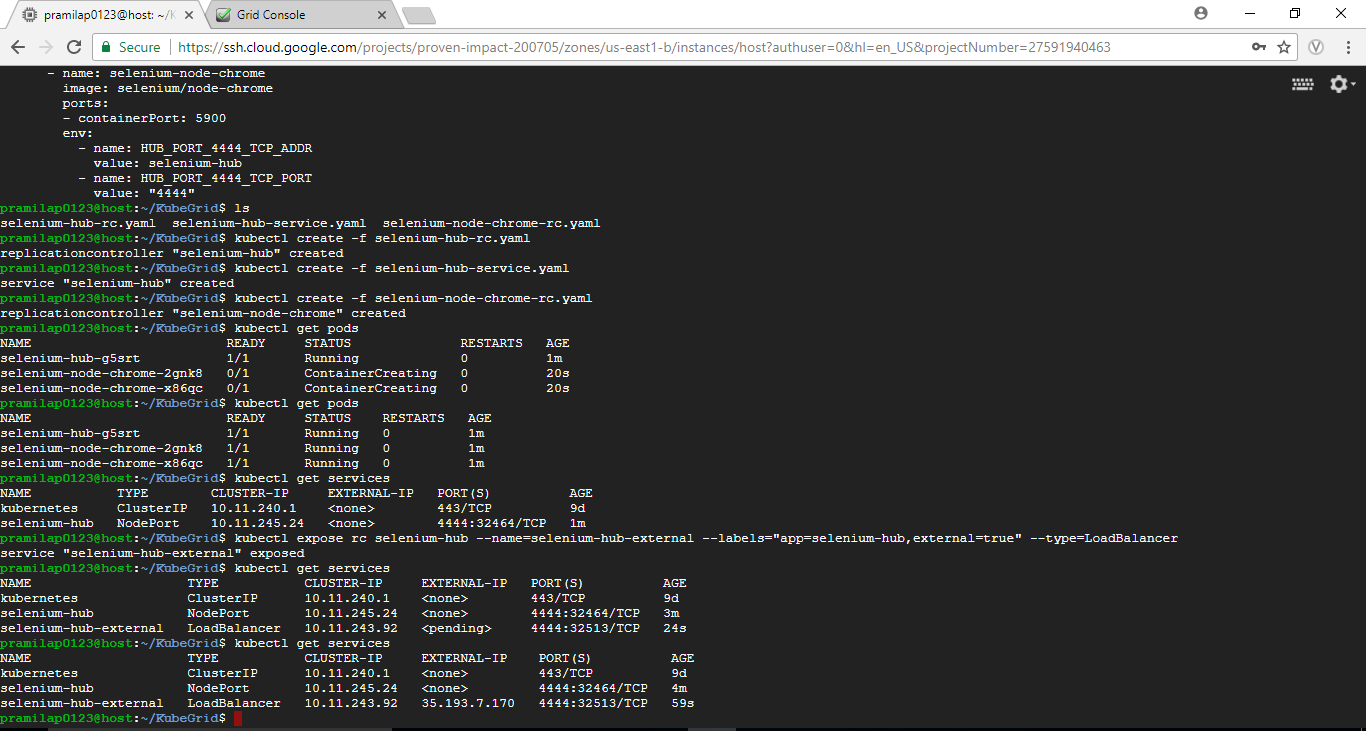
After creating all the files, you can deploy it by using command as kubectl apply-f /kubeGrid or deploy each file individually as follows



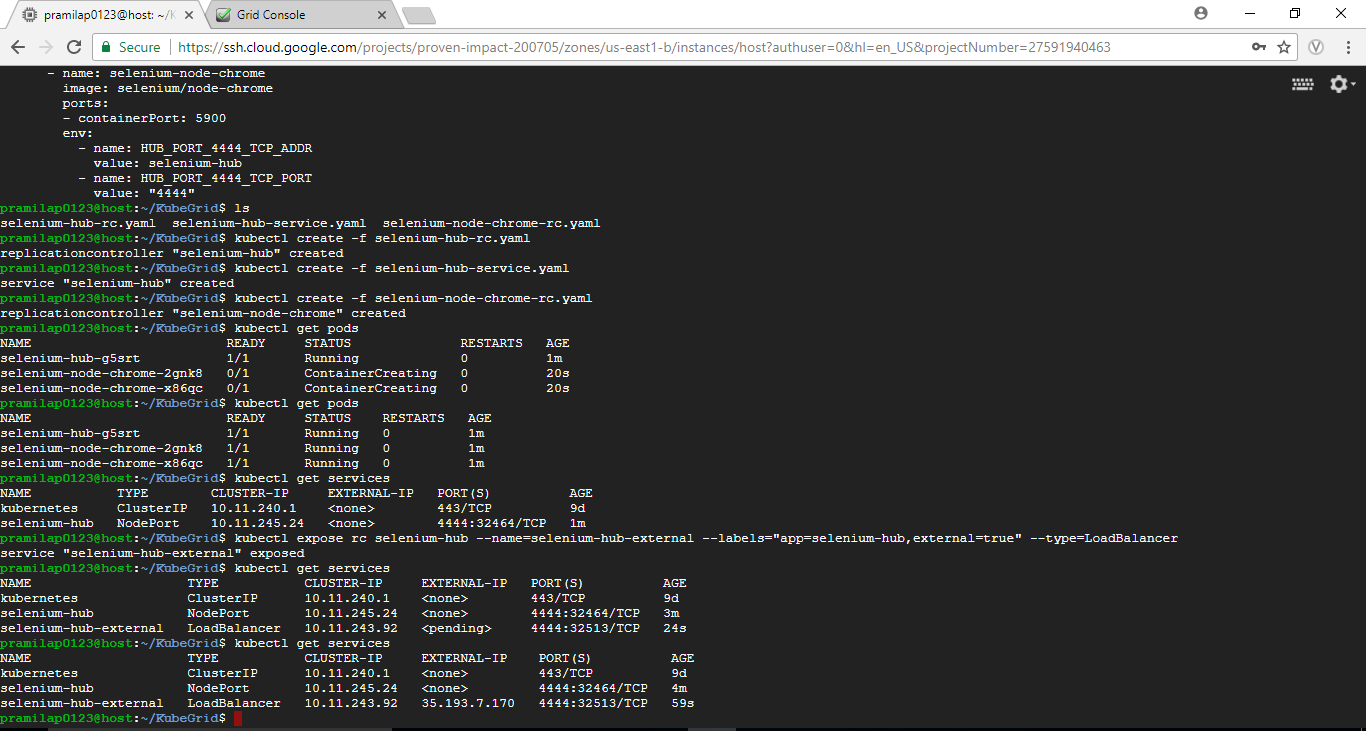
You can view the pods created for our selenium hub after some time as below



Now we can access selenium hub internally but to expose the selenium hub for others to access for executing their testcases, kubectl expose command can be used as follows



After exposing the selenium hub, its external ip address is generated which can be used to access it.



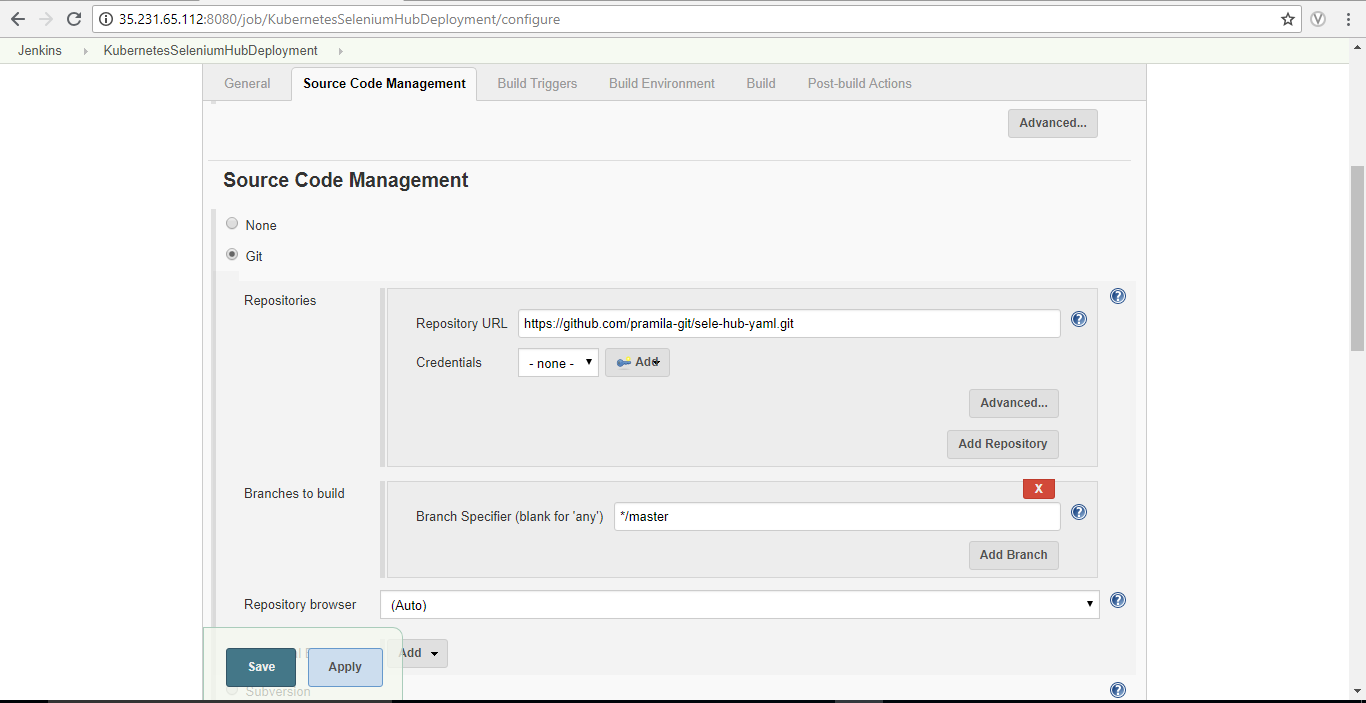
So you can view selenium hub console through <http://external-ip:4444/gird/console>

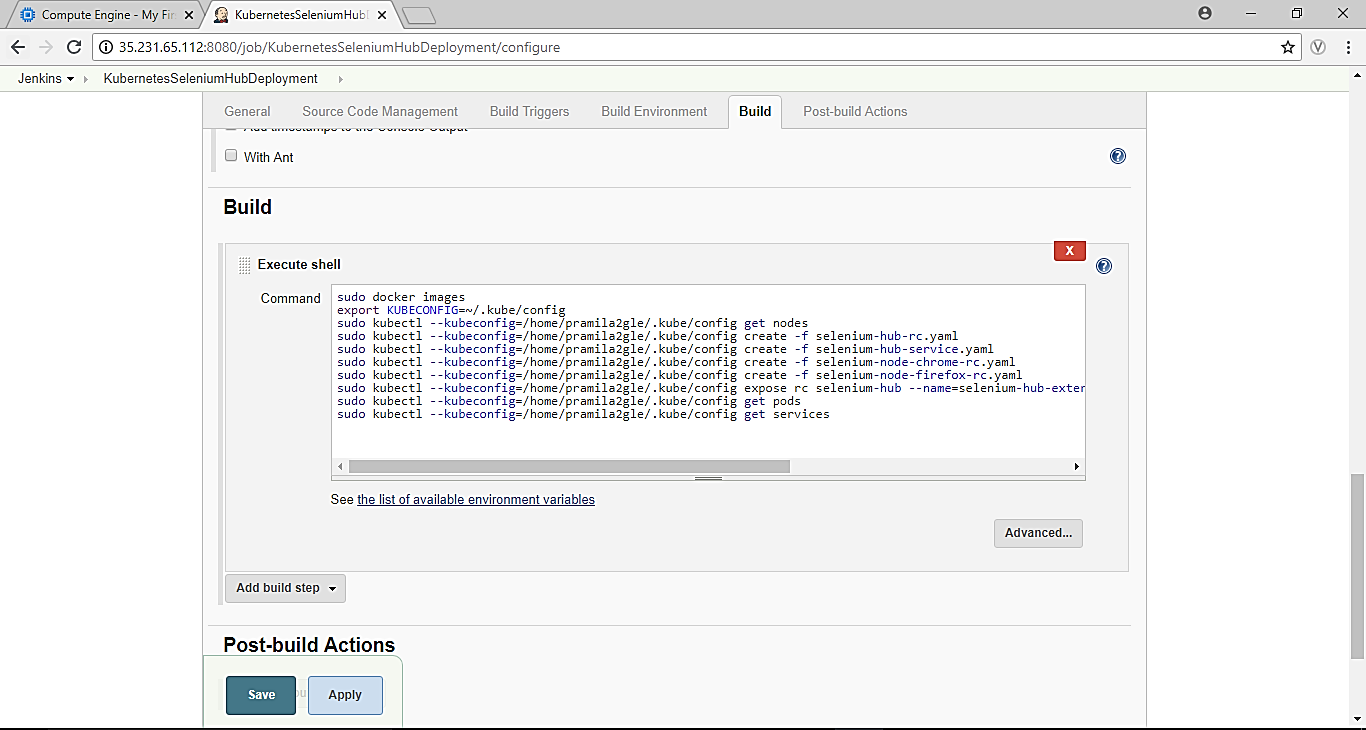


Now, anyone can execute their testcases in this selenium hub by mentioning the external ip in their script.

3.2.2 Through Jenkins job

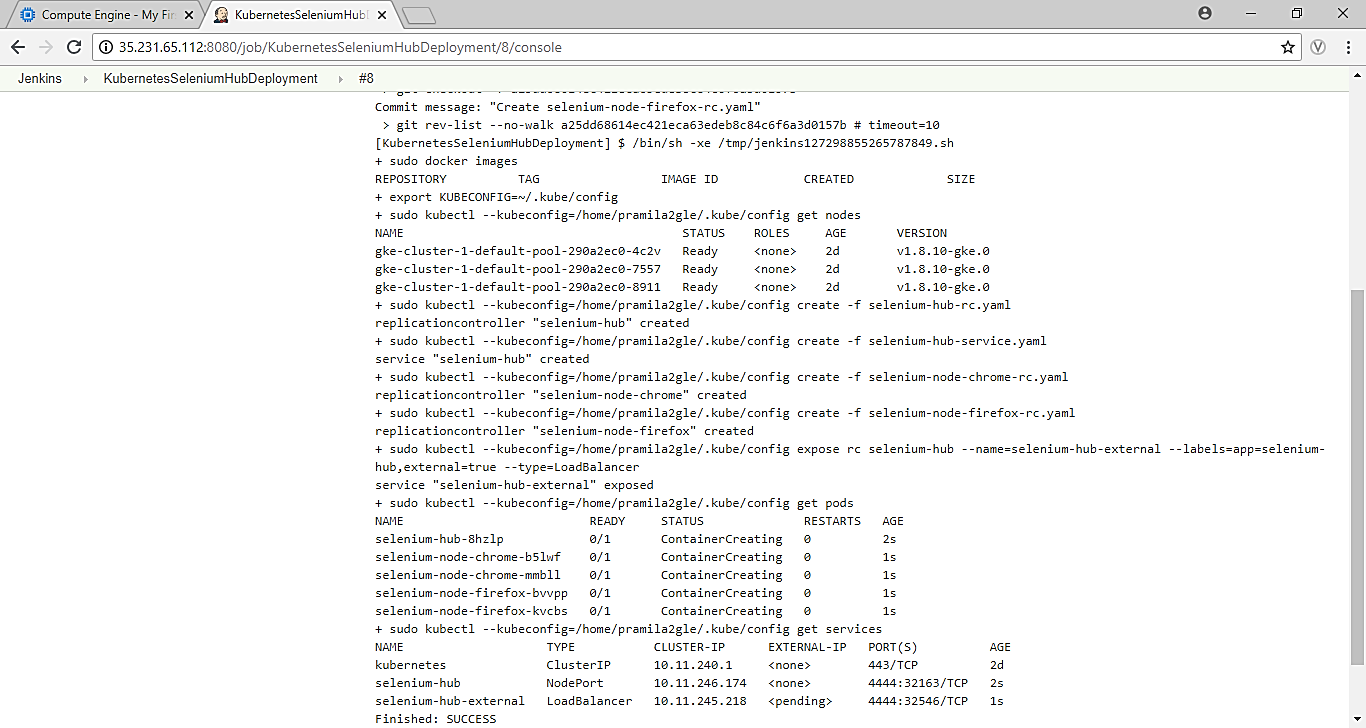
Create a freestyle Jenkins job which would retrieve YAML files from Git and shell script as follows:





Here, we are providing the path of kubernetes config file(replace with your path) using option --kubeconfig for every kubectl commands as these action should be done on the cluster which we have linked earlier with our VM.

Its output will be as follows after building the job.



So, now the selenium grid is deployed in the kubernetes cluster through jenkins job.

1. Deploying zalenium in kubernetes cluster

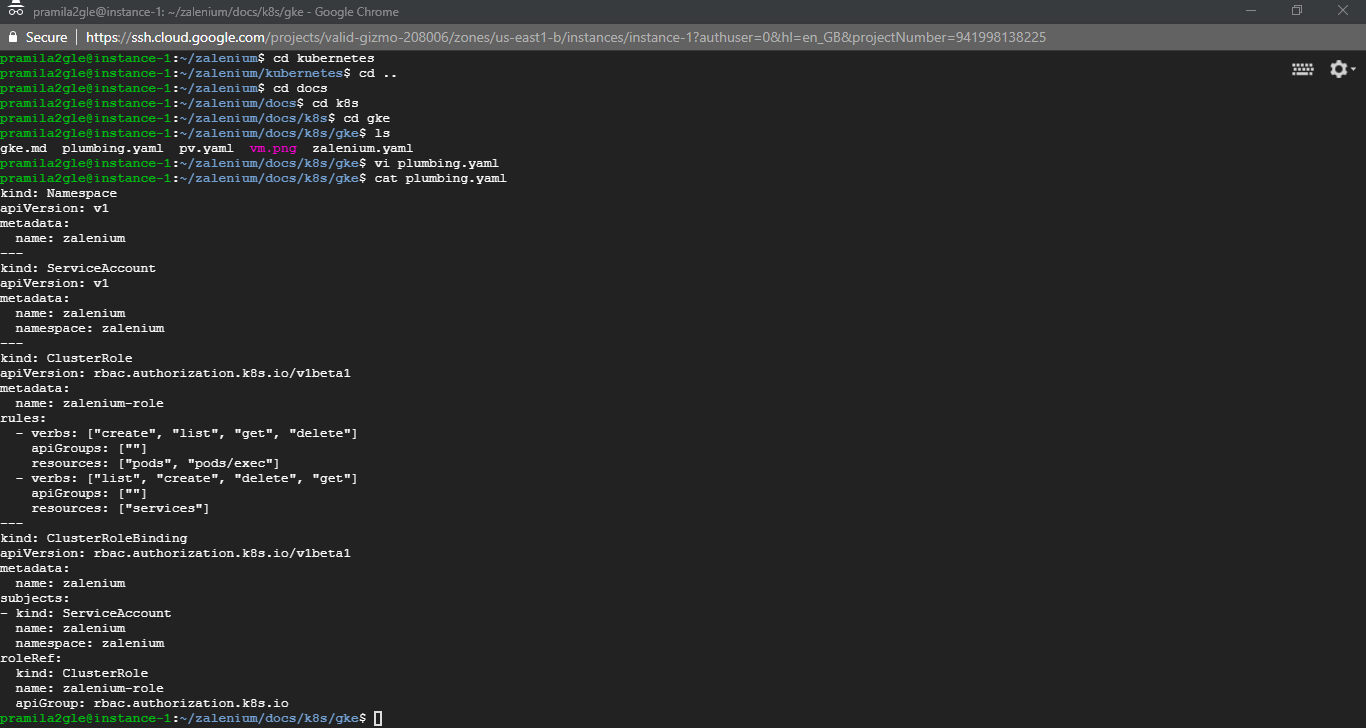
Zalenium is an extension of selenium grid where we need not use different docker images like we did in the above process for selenium/chrome and selenium firefox, as zalenium docker image itself internally contains selenium grid that contains one chrome and firefox node.

4.1 Creating YAML files for deployment

We have to create 3 YAML files for zalenium deployment: plumbing.yaml, pv.yaml and zalenium.yaml. Create YAML files by following the zalenium official gihub repository <https://github.com/zalando/zalenium/blob/master/docs/k8s/gke/>

Also, YAML files are available in the gihub repository <https://github.com/pramila-git/zalenium-yaml/>

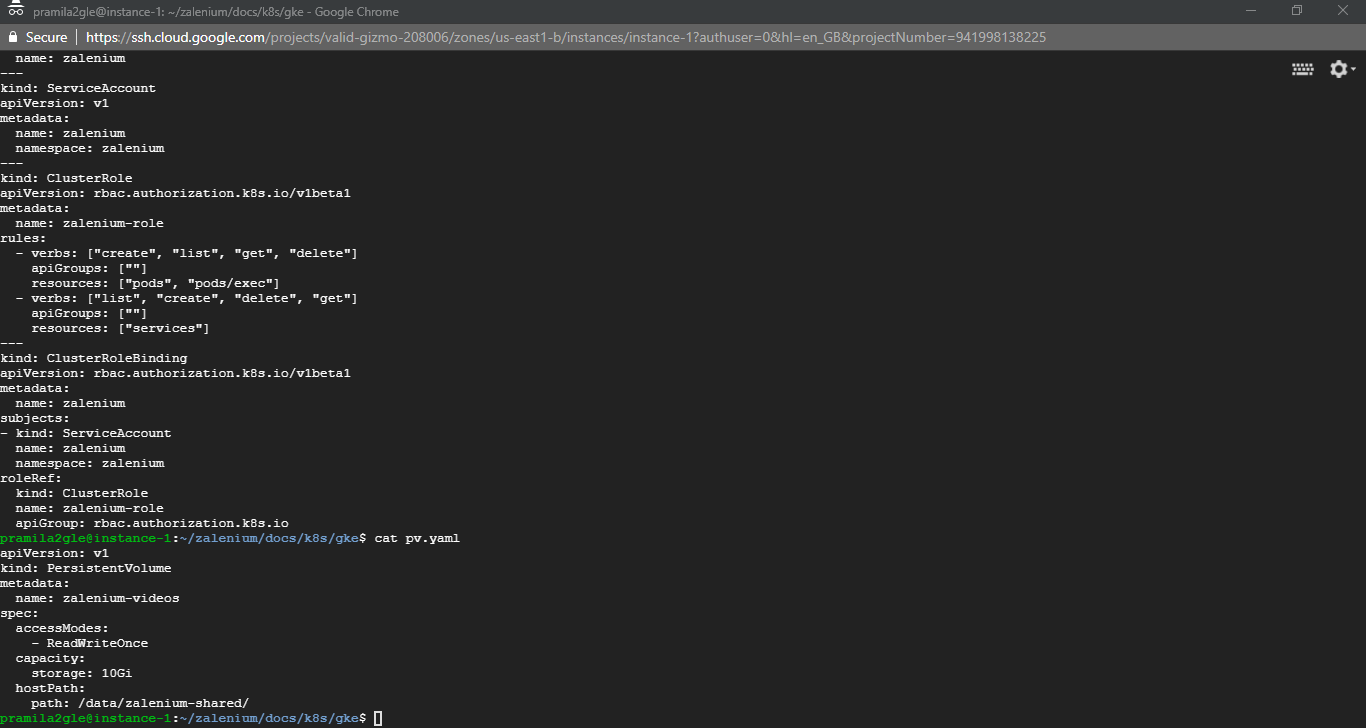
YAML file for plumbing.yaml file



In this YAML file, it will create a different namespace “zalenium” where our pods and services will be deployed instead of default one. Then, it will create service account to more access.

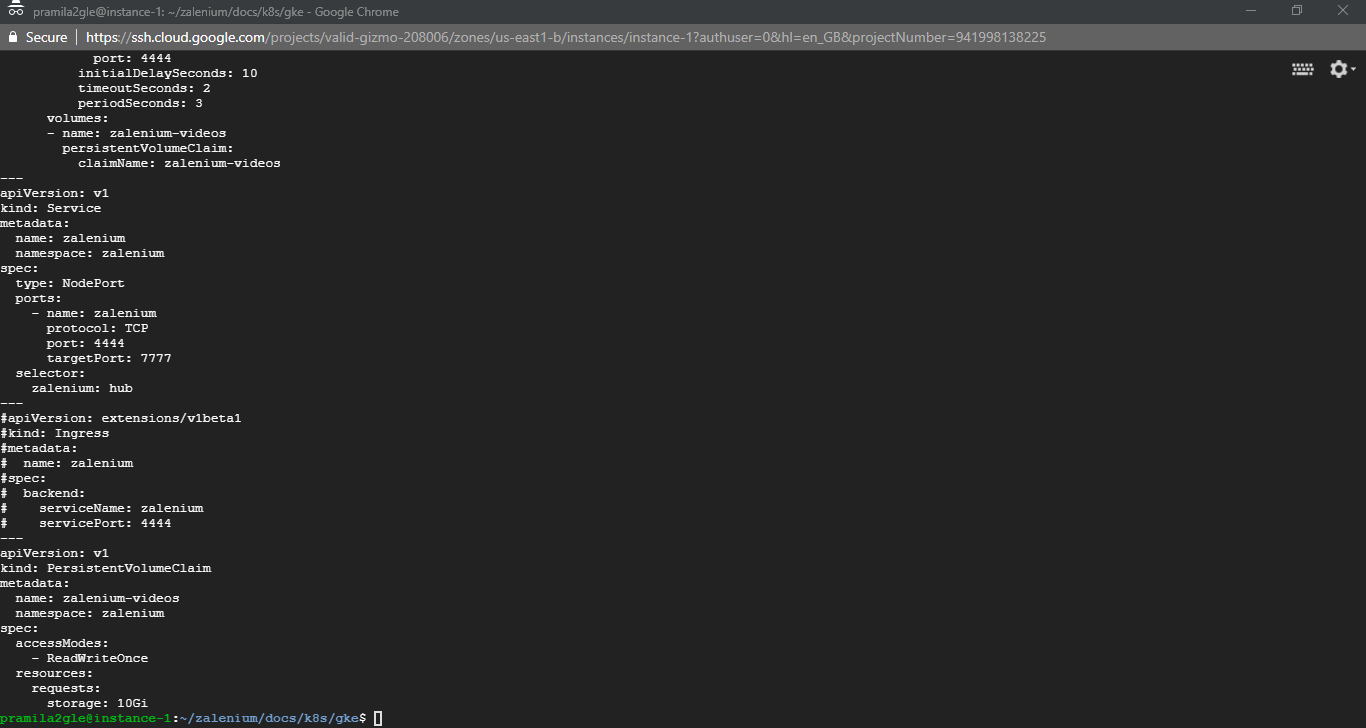
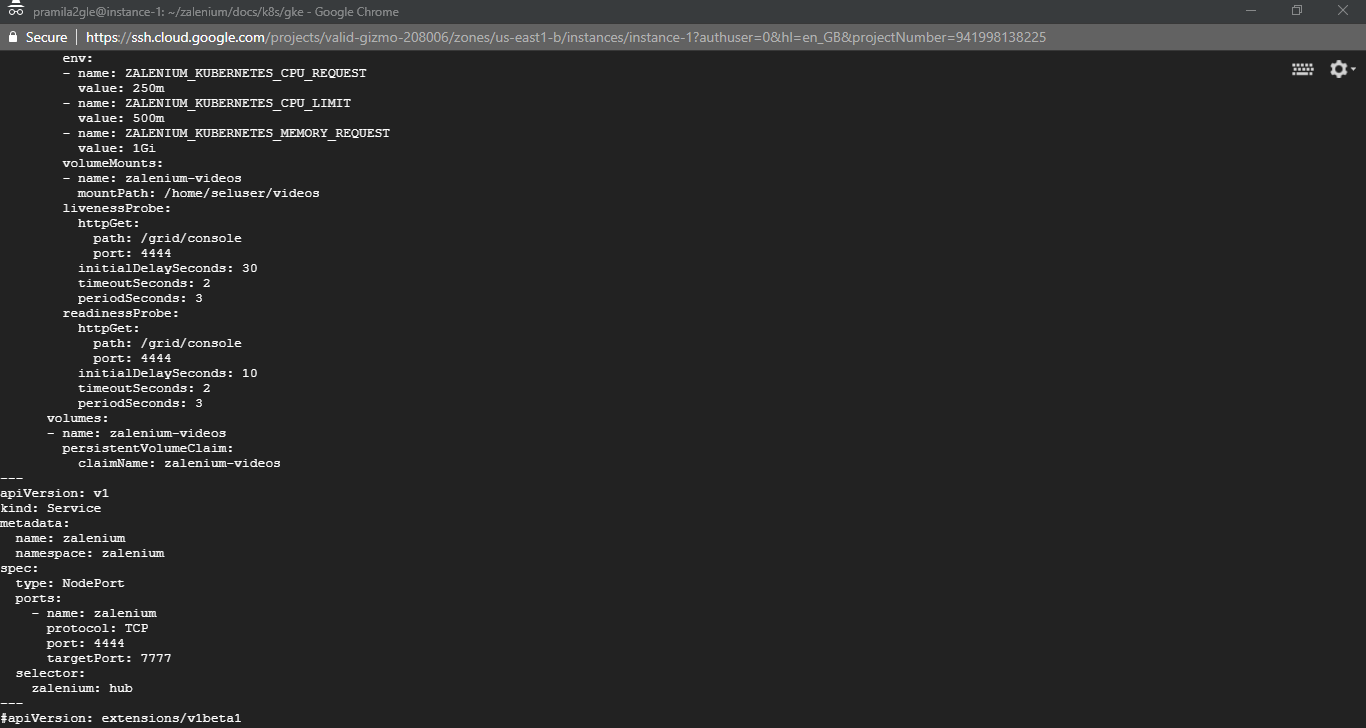
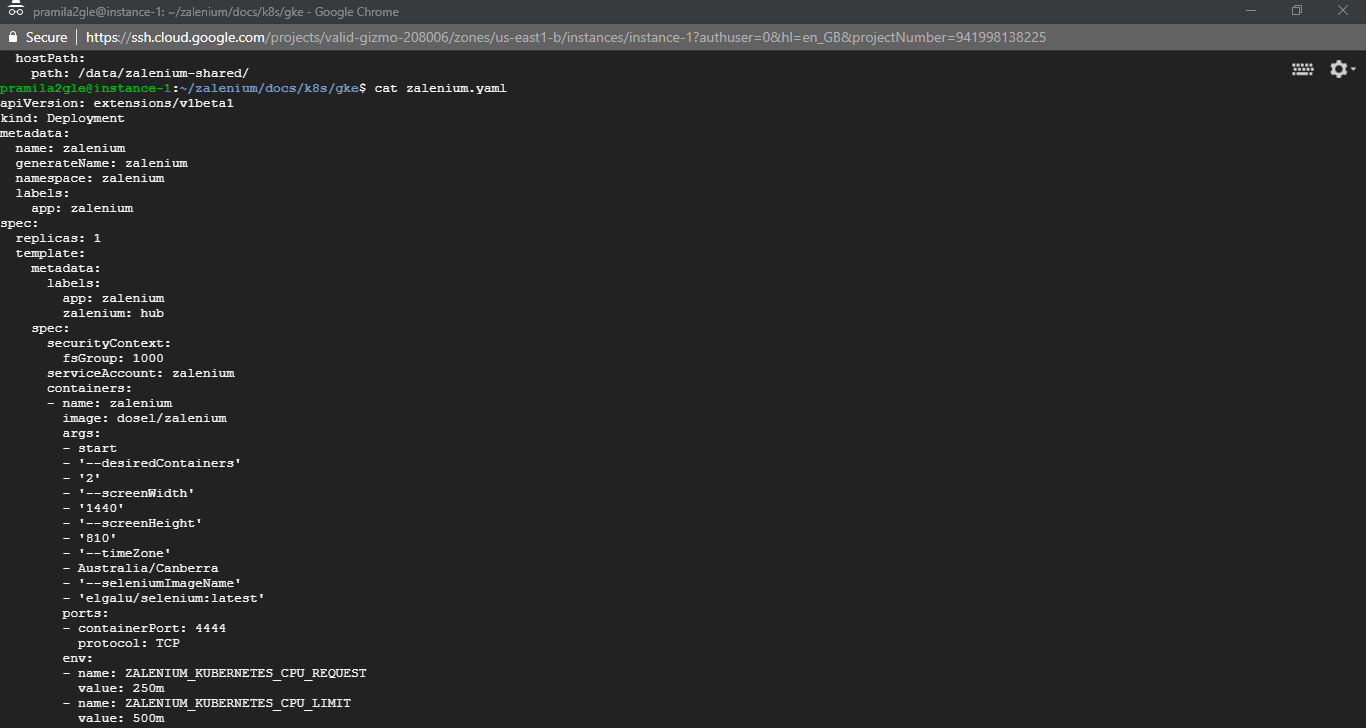
Also, cluster roles are defined so that it can create and destroy the containers on demand with the help of service account.

YAML file for pv.yaml file



This YAML file will create a permanent volume to store the test case videos captured by zalenium in the specified path.

YAML file for zalenium.yaml

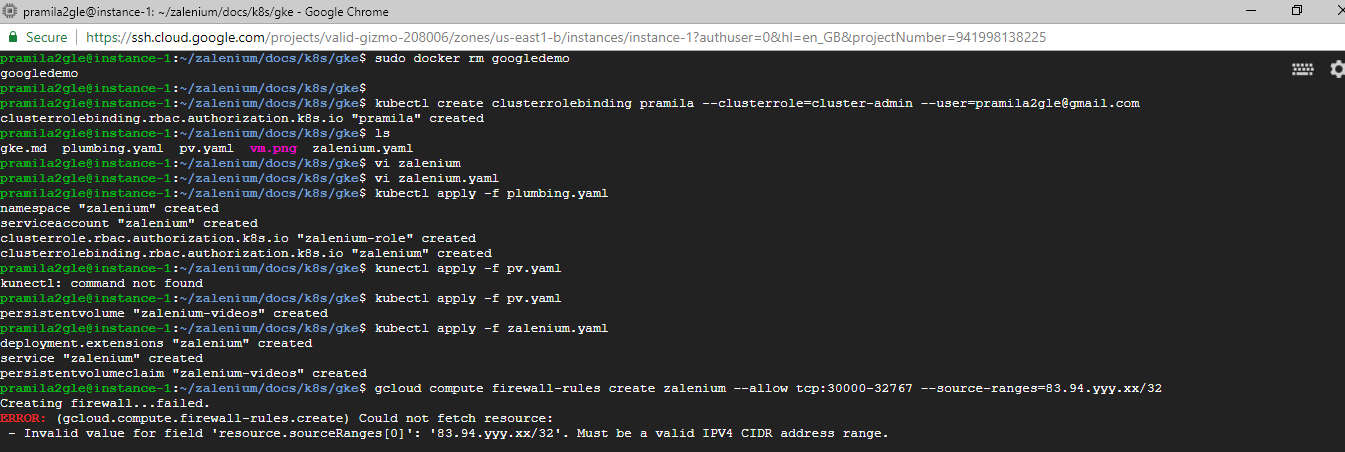


This YAML file creates deploys the zalenium container with the specified parameters and created a service of it. Also, it defines the volume mounting of the captures logs and videos which will be linked with the persistent volume defined in pv.yaml file.

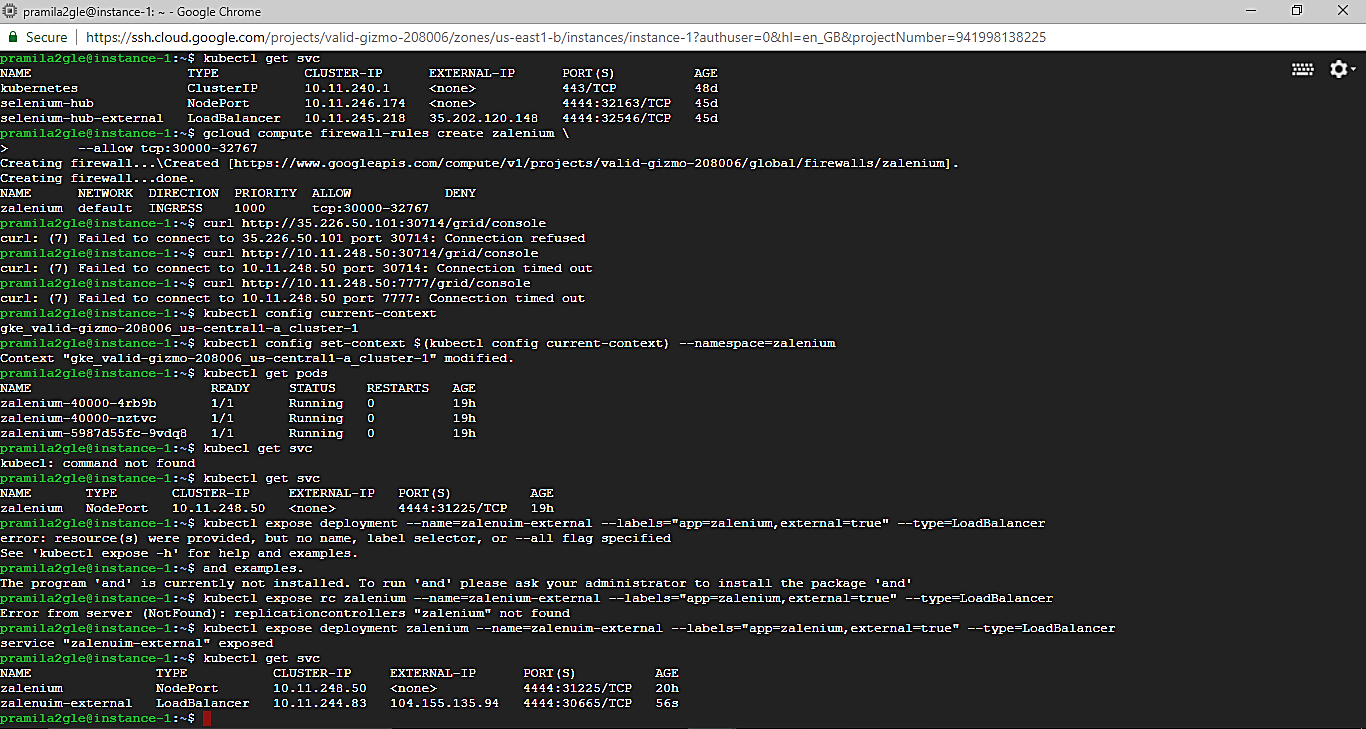
4.2 Deploy YAML files

4.2.1 Through commands

Before deploying kubernetes cluster, we have to execute below command to provide zalenium with the permission to create containers as per requirements.

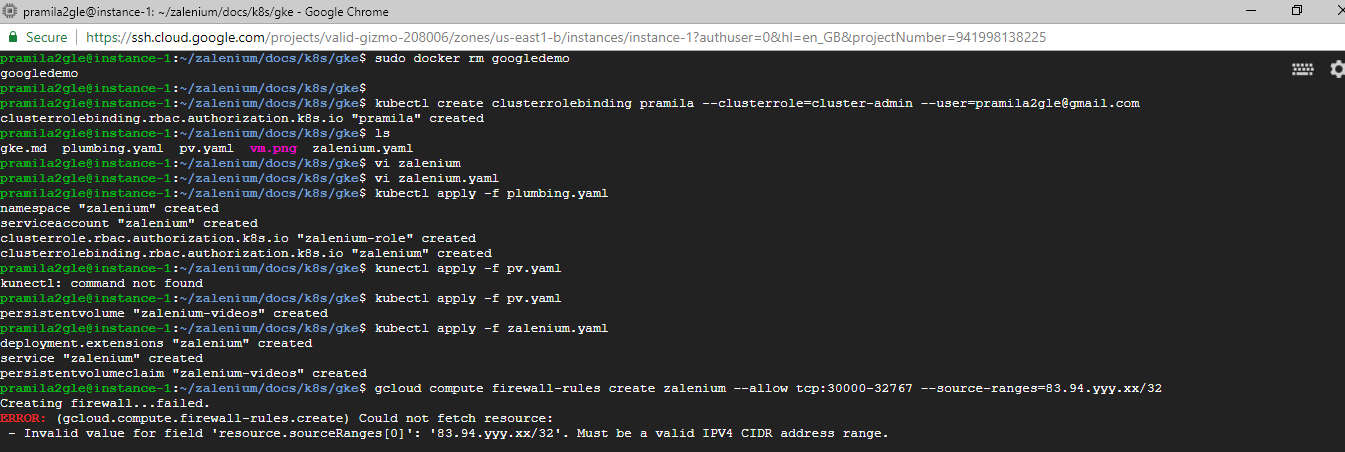
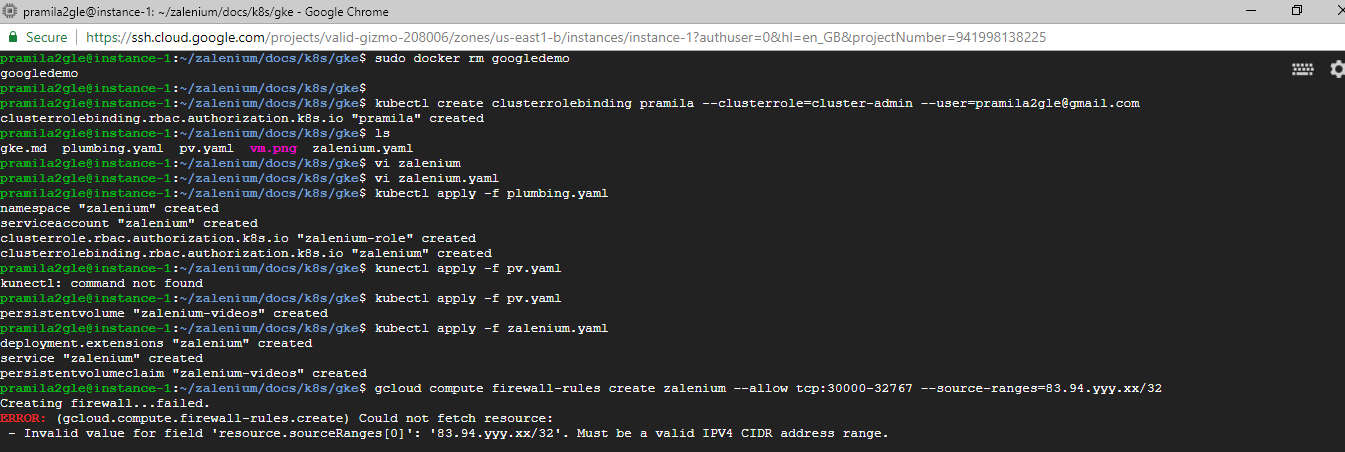


Also, set firewall ruled in GCP for zalenium through the following command:



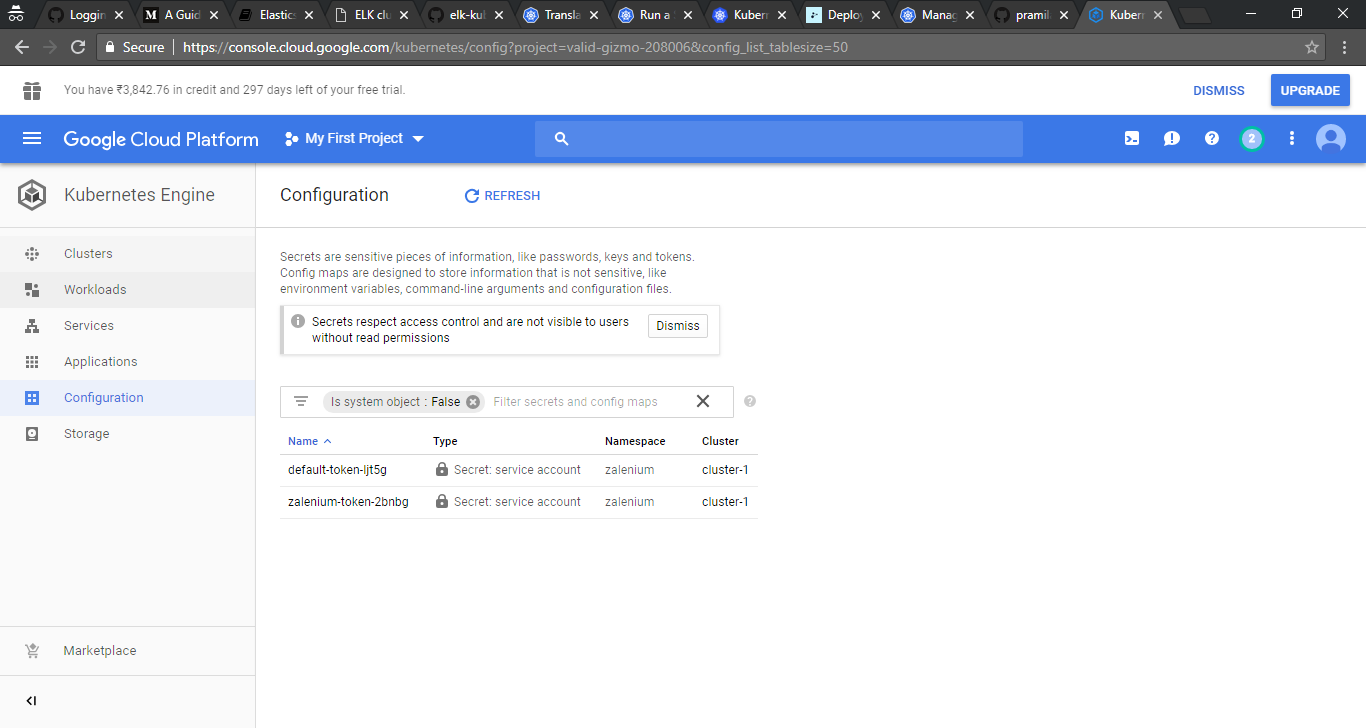
Just replace the gmail id with yours and can also give any name instead of “pramila”

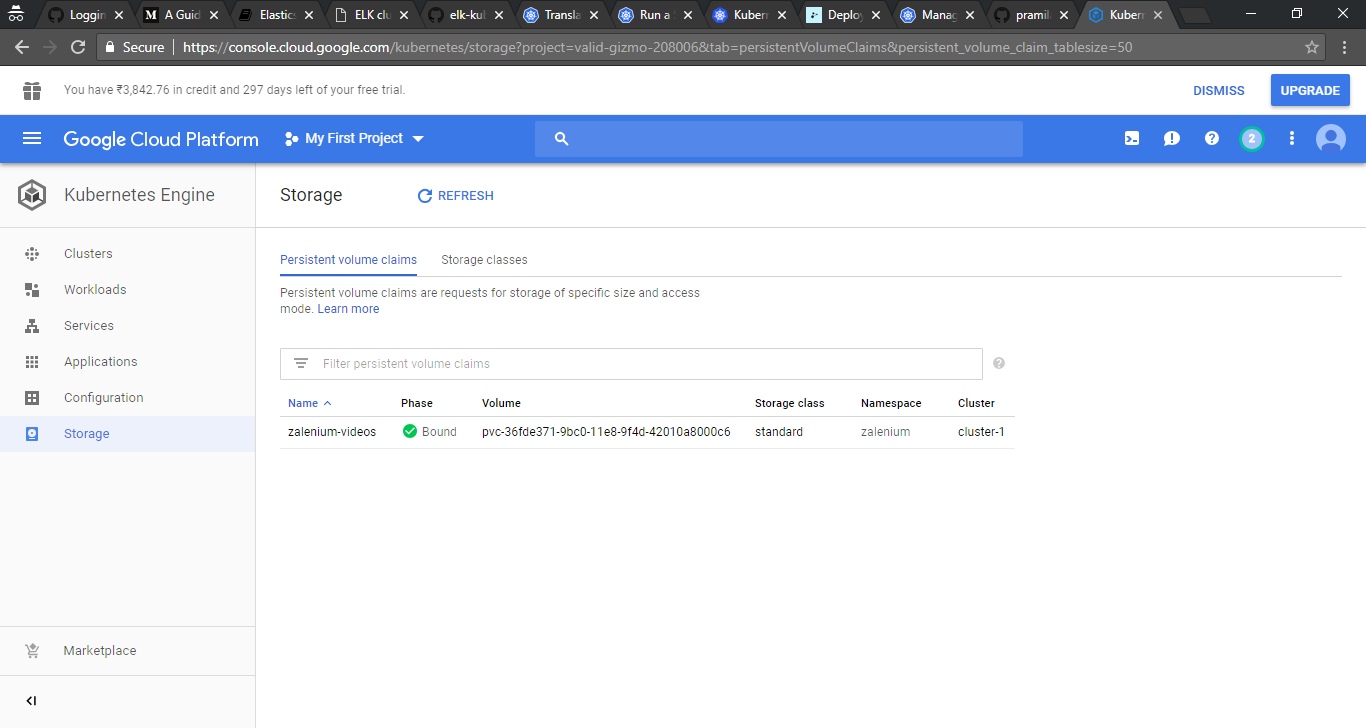
Now deploy zalenium in kubernetes cluster as follows after retrieving the YAML files from the Github repository.



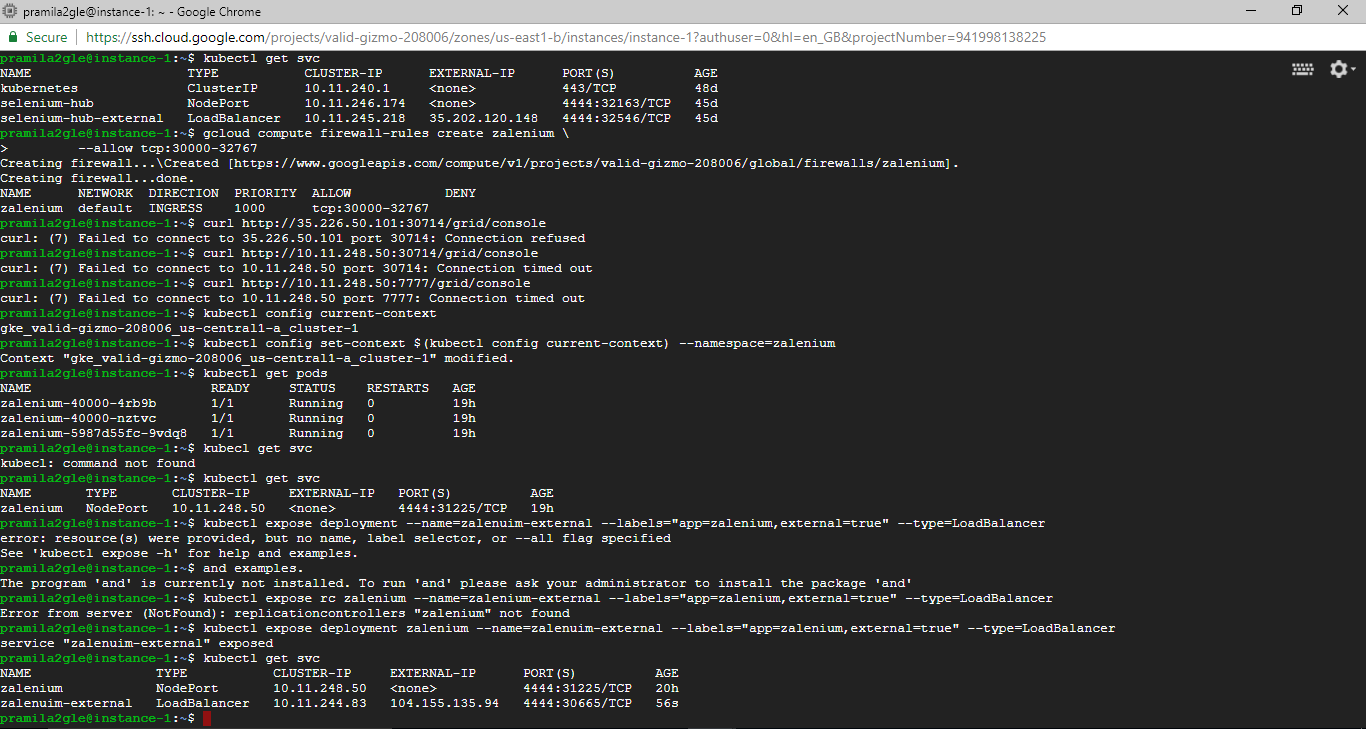
As soon as zalenium is deployed, we will see the following changes in GCP kubernetes dashboard

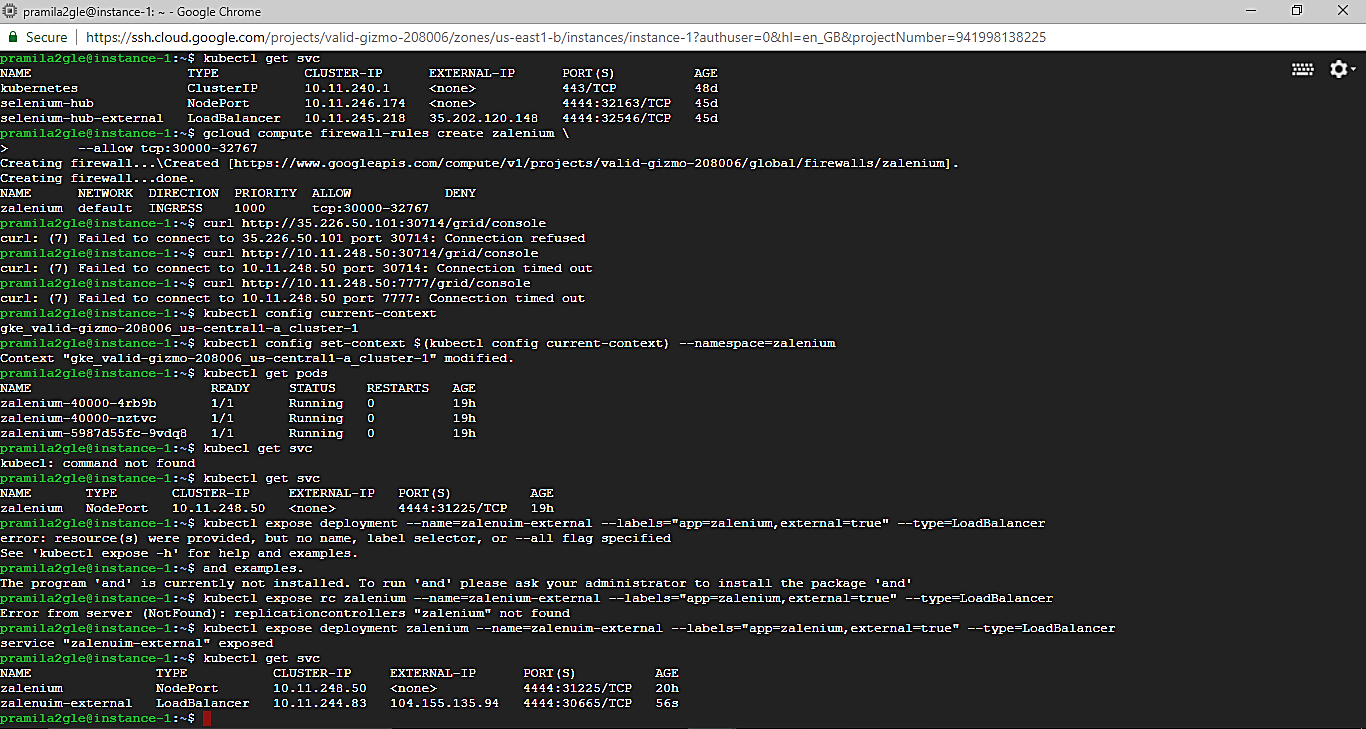
Zaleium pods, its service, token for service account and persistent volume for storing zaleium dashbaord videso are created.



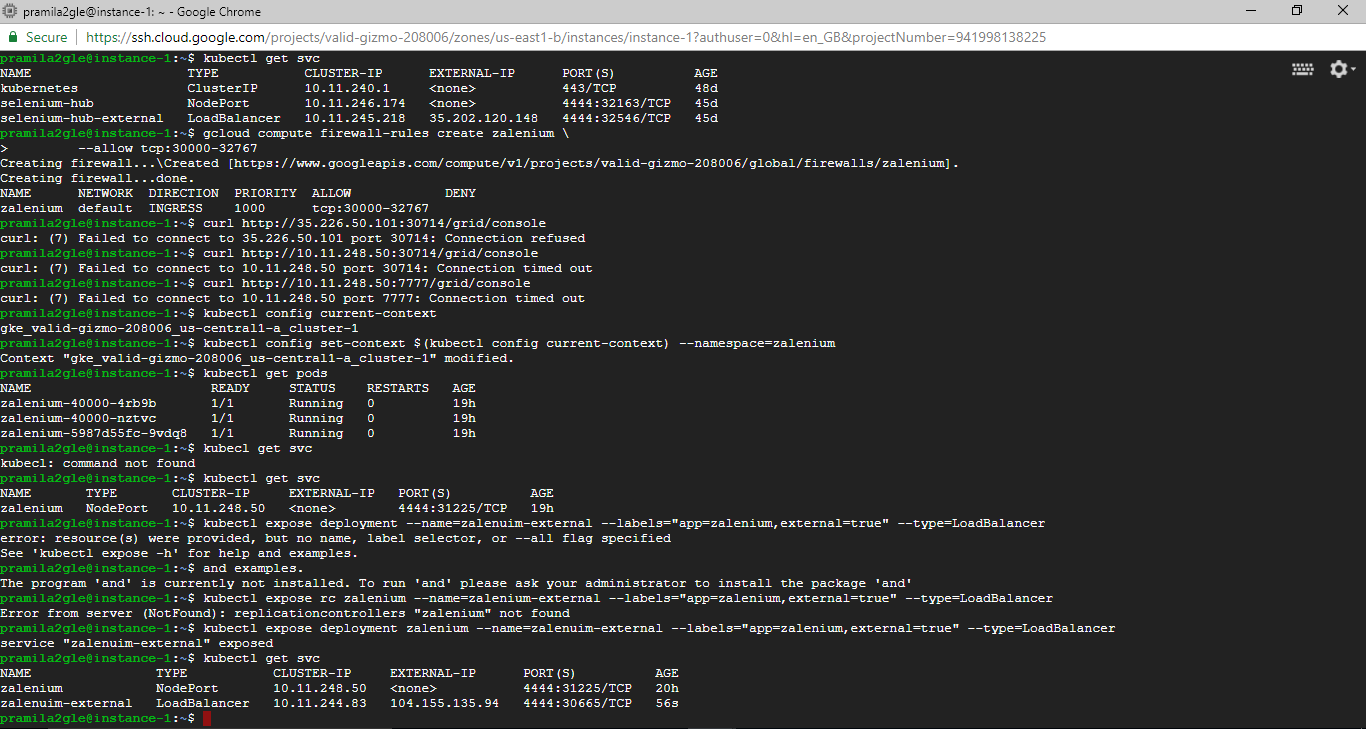


To see the pods and services in the controller VM, we have to switch to zalenium namespace as follows:





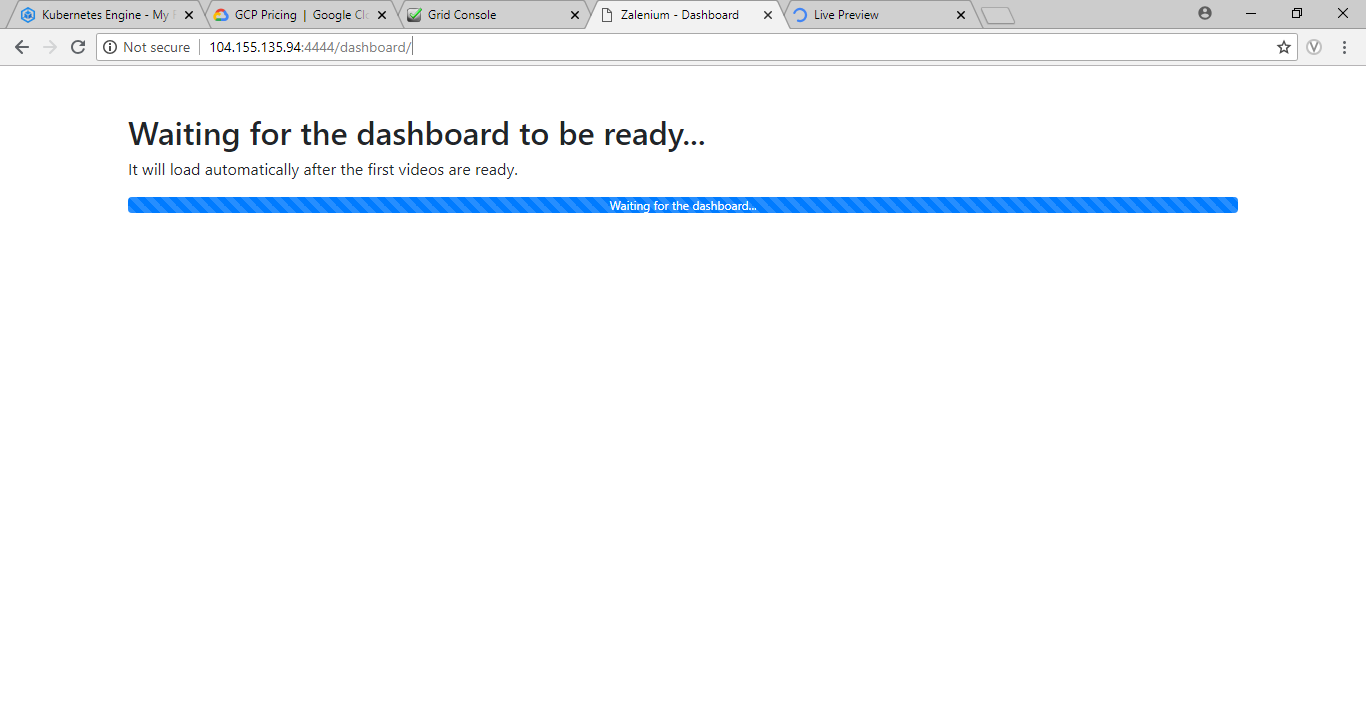
We will now expose the zalenium service to access it in from browser as follows

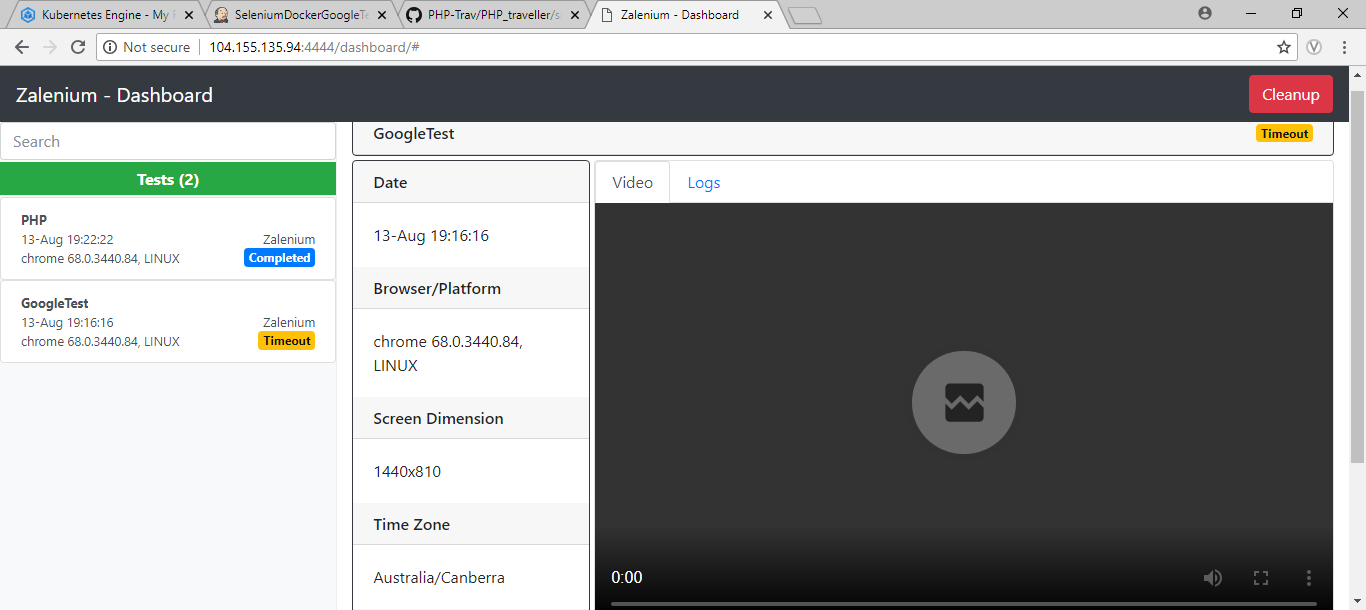


So, now, we can access selenium grd through http://<<external\_ip>>:4444/grid/console

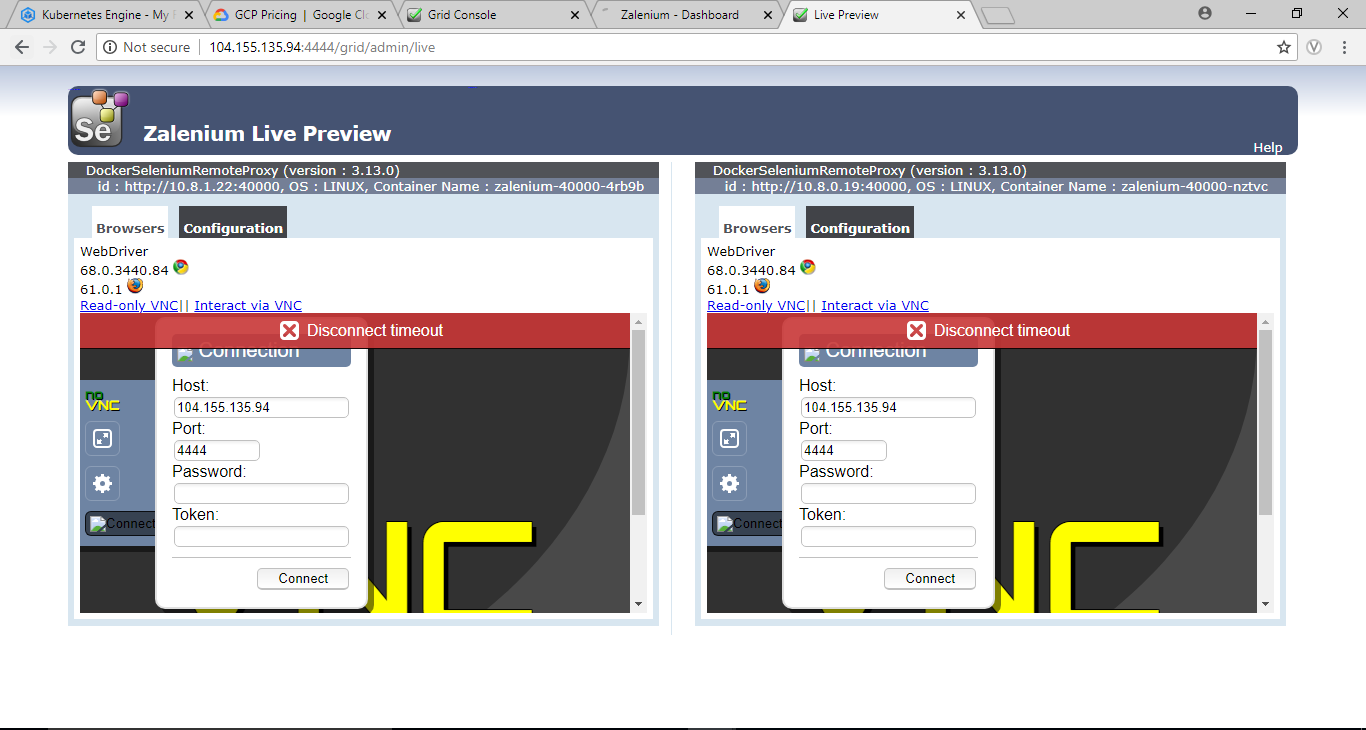


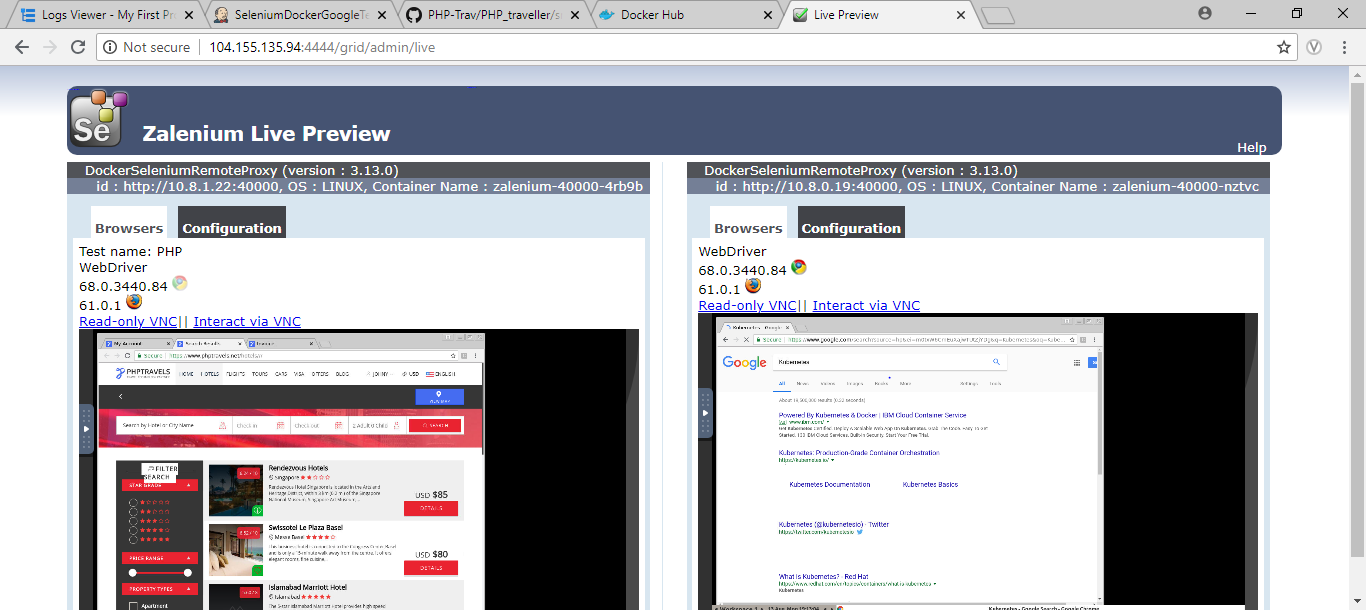
Also, we can access zalenium’s dashboard through the link http://<<external\_ip:4444>>/dashboard where the videos & information of testcases executed in the selenium grid will be shown as soon as testcases are executed in the selenium grid .





We can see the testcases executing in the browser nodes lively through the link http://<<external\_ip>>:4444/admin/live





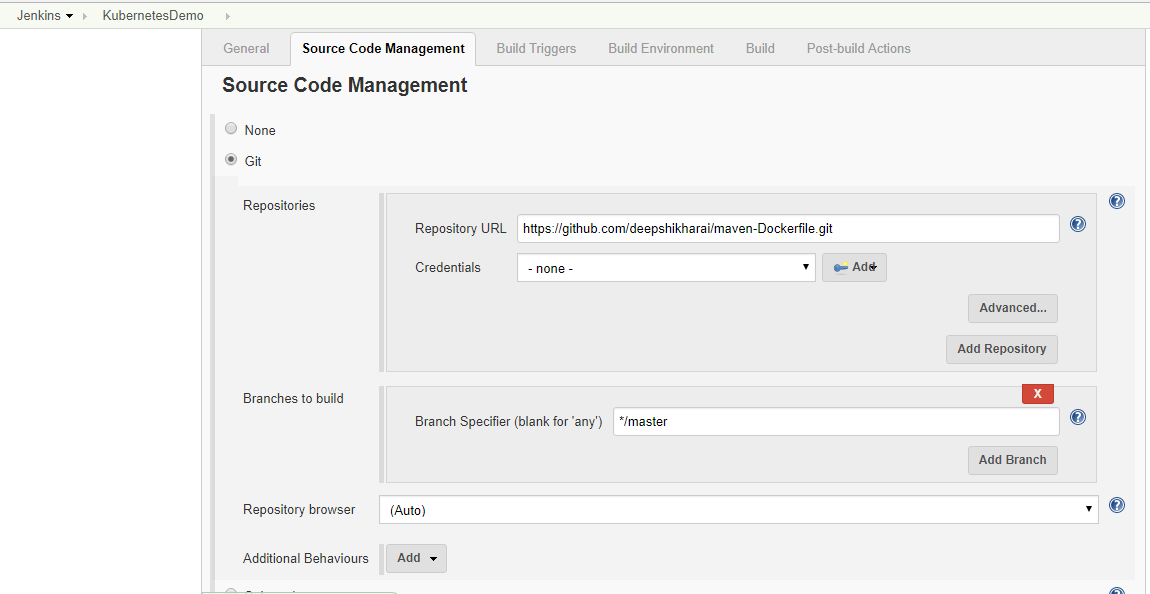
So, now we can run our testcases in zalenium by providing its external service link similar to selenium grid.

5. Jenkins job creation for docker image building

You can install Jenkins in the VM by referring document.

We have already created a docker image of simple google testcase manually. We will automate the process of building already created dockerfile from git and push it in the docker hub through Jenkins.

Create a freestyle Jenkins job. Configure it by adding git repository as



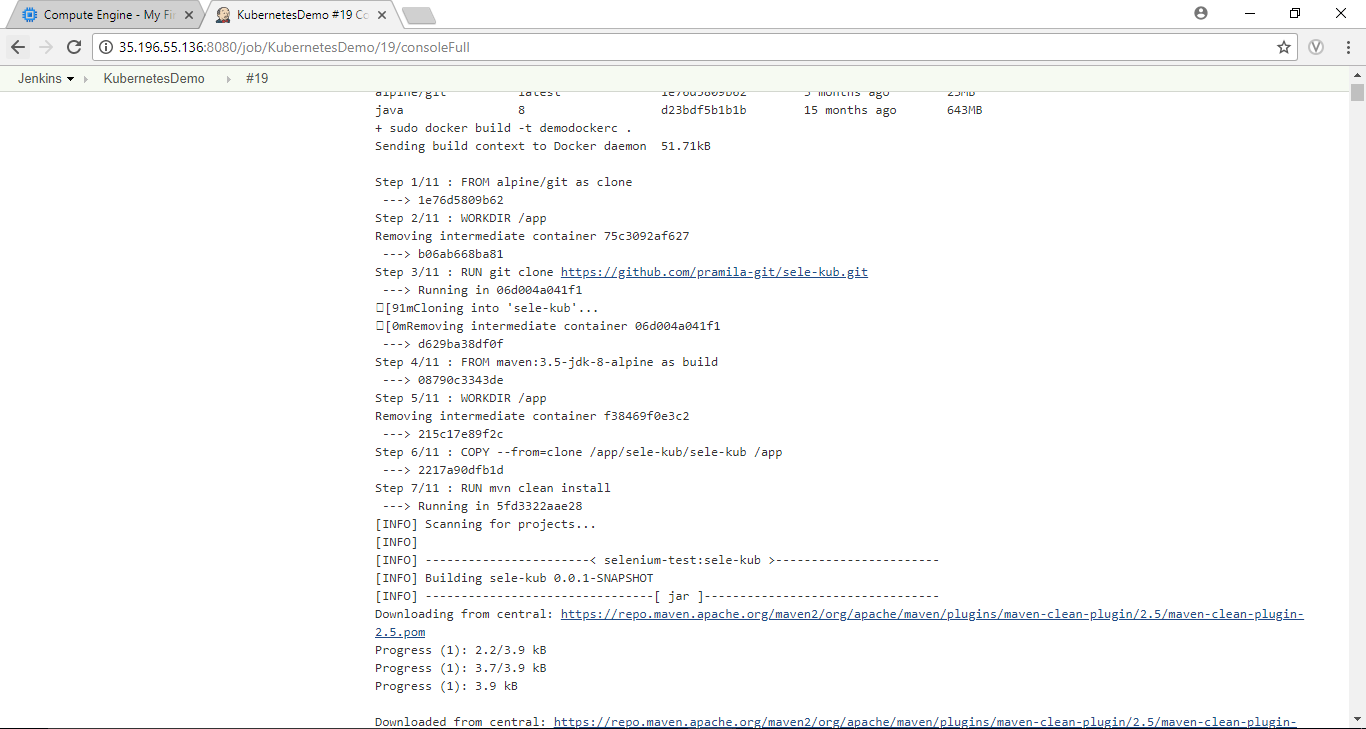
Then add build step as execute shell with following contents

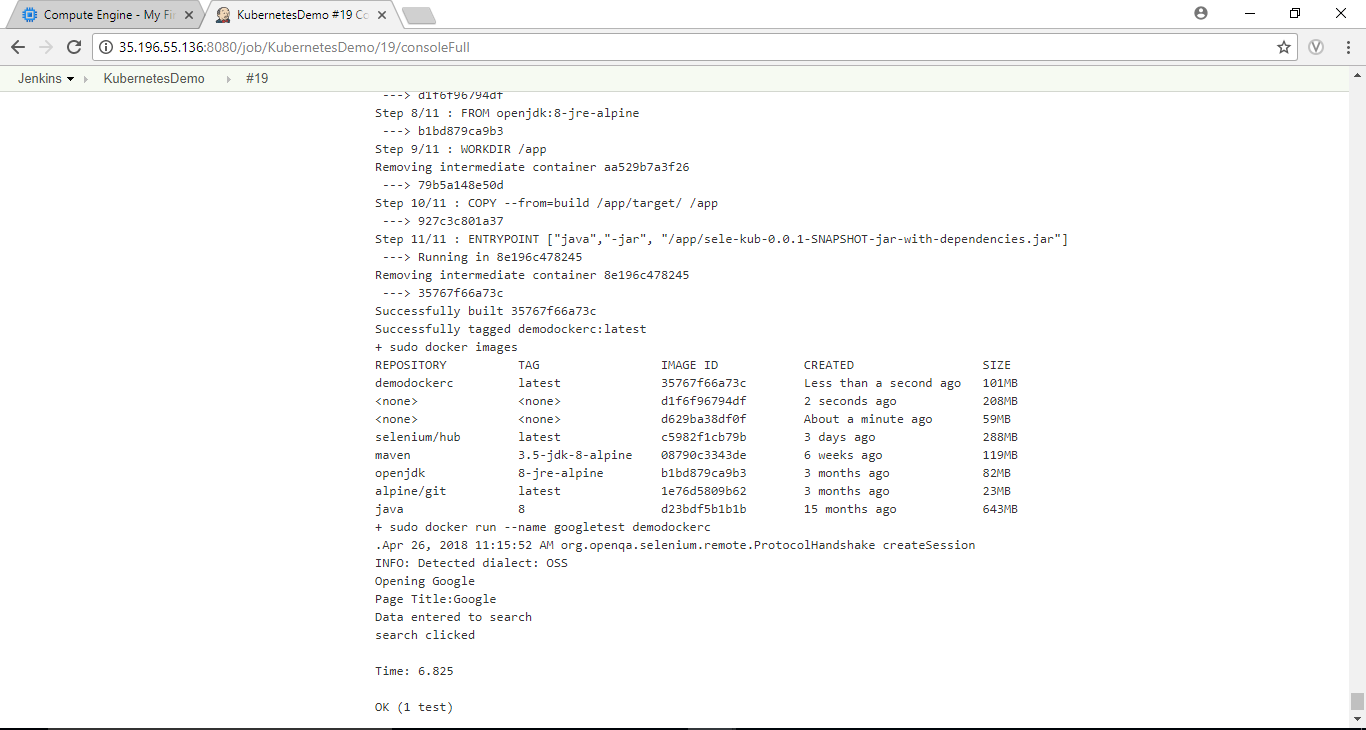


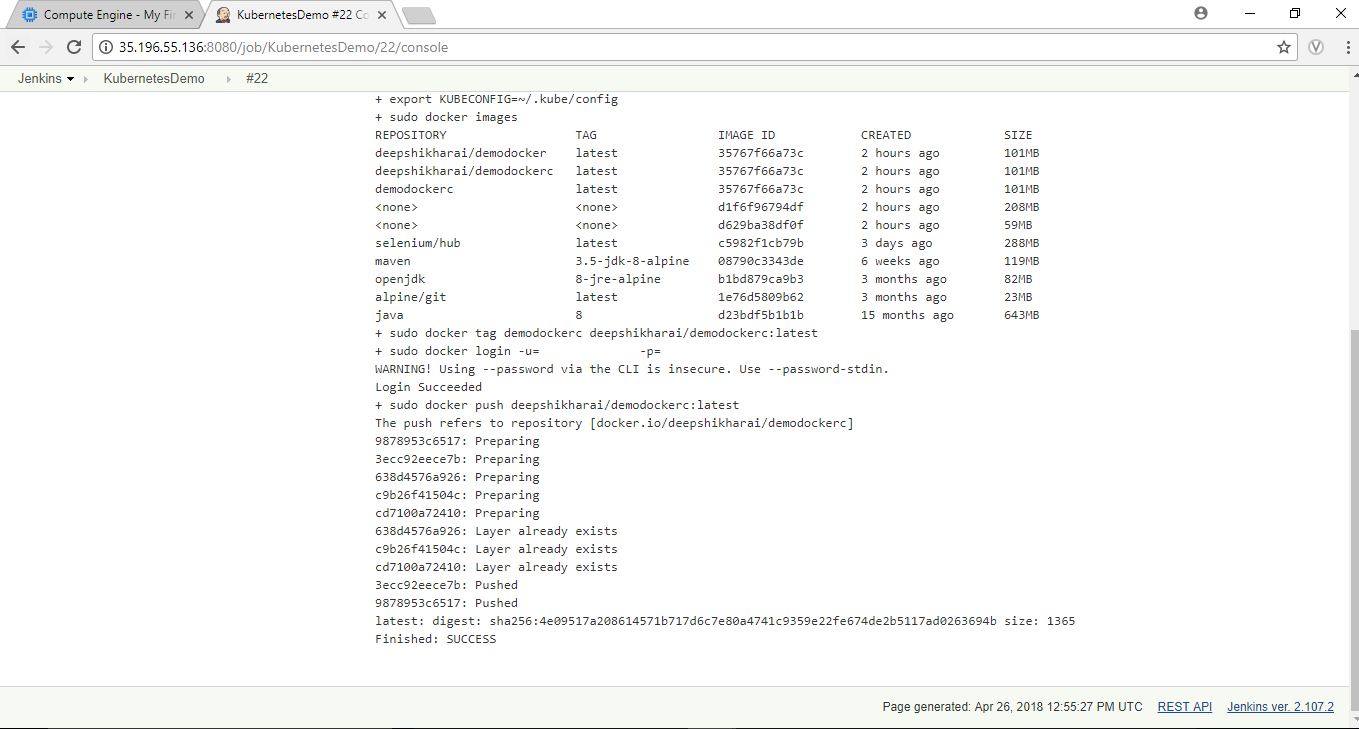
In this shell script, we are

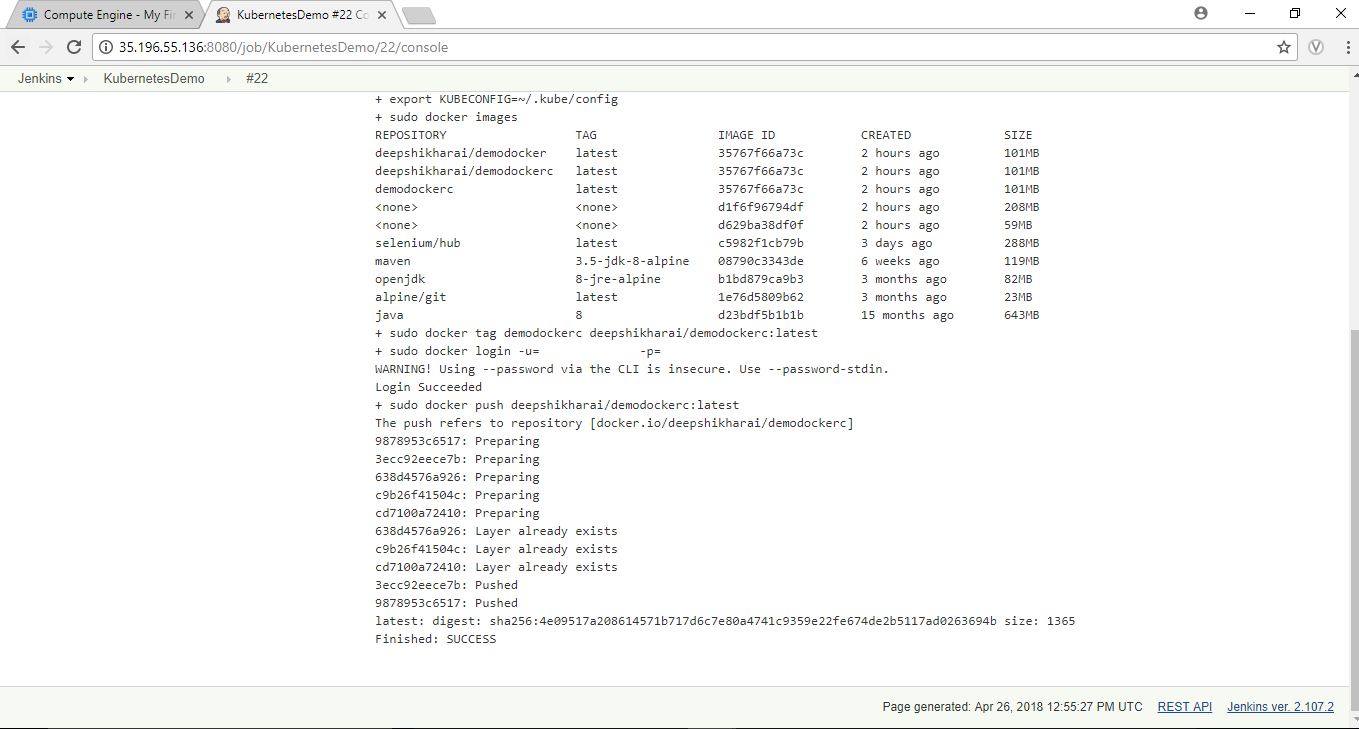
* Building the dockerfile
* Running the docker image created from the build
* Tagging the docker for pushing it into deepshikhrai docker hub account
* Pushing the docker image into the docker hub

You will output as below after configuring the Jenkins job and building it







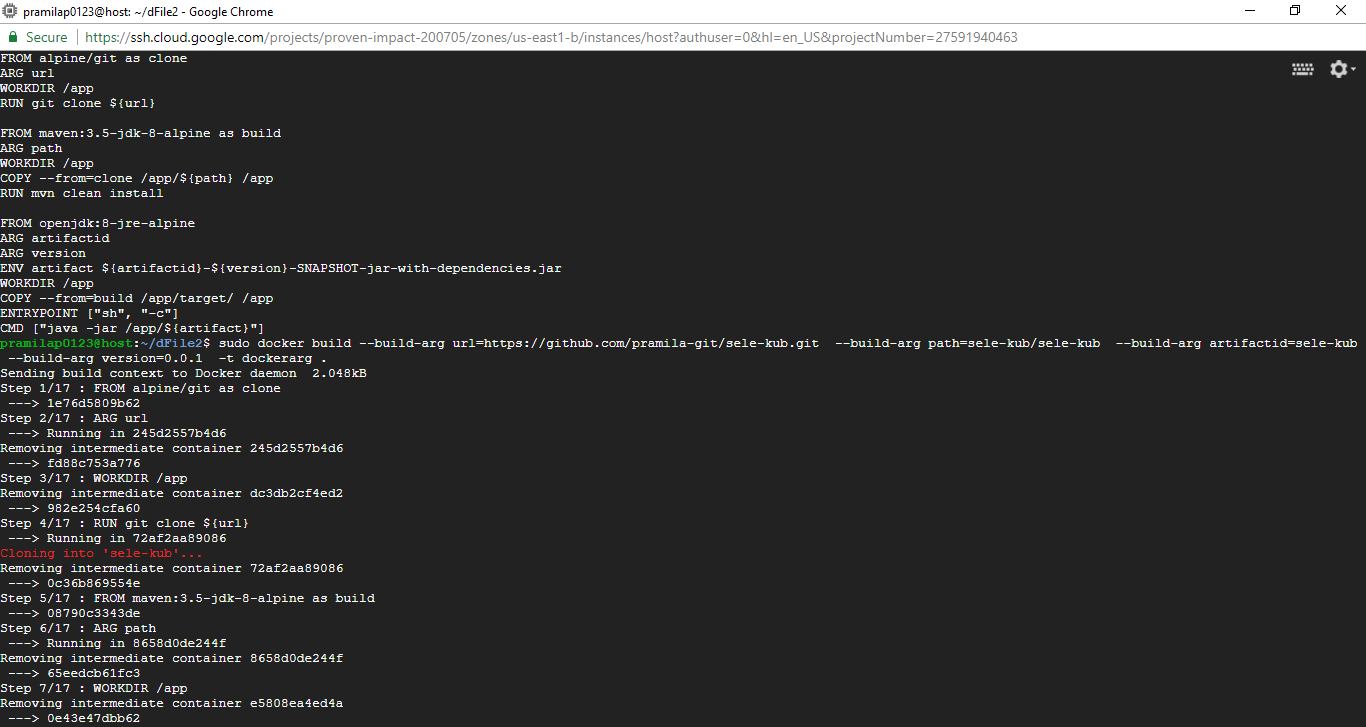


1. Creating dockerfile with arguments

In the dockerfile created previously, we have hardcoded the values inside the dockerfile.

So will parametrize it by passing the values like git url, project name & version arguments while building the dockerfile.

So, create dockerfile inside a folder with the following contents



About each the line newly added in the dockerfile:

* In the 2nd line, we specify url as argument.
* In the 4th line, we are appending url argument’s value while git cloning.
* In the 6th line, we specify path as argument.
* In the 8th line, we are appending path argument’s value while navigating to the project’s path.
* In the 11th & 12th line, we specify artifactid & version as arguments.
* In the 13th line, we are defining artifact as environment variable and defining its value by artifactid and version. So if artifactid is project and version is 0.0.1, then artifact value would be project-0.0.1-SNAPSHOT-jar-with-dependencies.jar which jarfile name of the project.
* In the 16th line, Entrypoint is used so that the CMD command will be executed whenever its docker image is run.
* In the last line, we are appending the artifact argument’s value to run the jar file.

Now we can build this dockerfile by arguments after navigating to its folder as

docker build --build-arg url=<<git url>> \

--build-arg path=<<path>>\

--build-arg artifactid=<<project name>> \

--build-arg version=<<version number>>\

-t <<image-name>> .

For example,

docker build --build-arg url=https://github.com/pramila-git/sele-kub.git\

--build-arg path=sele-kub/sele-kub\

--build-arg artifactid=sele-kub\

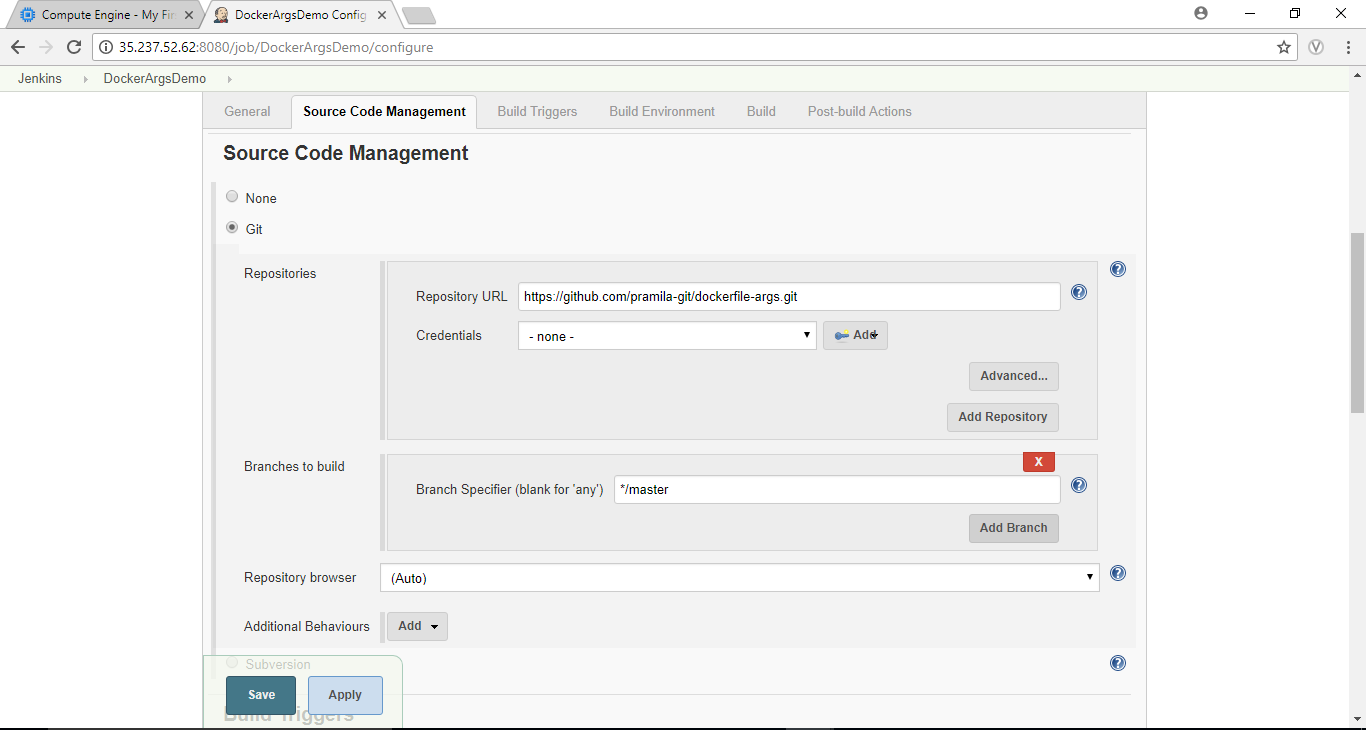
--build-arg version=0.0.1\

-t demo .

As soon you build the dockerfile, these arguments will be passed in the dockerfile and accordingly it will be executed.

1. Jenkins job creation for docker image building with parameters

We will create Jenkins job to run the dockerfile with arguments. Create a freestyle Jenkins job with configuration as follows:





You add tag and push commands as well at the end. Now build this job. Output is as follows:

