**TASK – IMPLEMENTATION OF CONTAINER ORCHESTRATION USING KUBERNETES ENGINE AND JENKINS PIPELINE**

**IN GOOGLE CLOUD PLATFORM**

**Step 1: Install Jenkins on VM Instance in GCP:**

Jenkins is a Java based software which can be installed from the Ubuntu packages. Jenkins is mainly used for Continues Integration and Continuous Deployment (CI CD).

**Prerequisites**

* A running Compute Engine.
* Installed Java 8.

Choose a VM Instance with at least 1 GB RAM

**Setup Firewall Rules**

Jenkins uses a custom port 8080 to run, so you need to create a firewall to all access to this port.

Go to **VPC Network >> Firewall rules** and click Create Firewall rules.

In **Name** enter Jenkins

In **Targets** select All instances in the network

In **Source filter** select IP ranges

In **Source IP ranges** enter 0.0.0.0/0

In **Protocols and ports** check **TCP** and enter 8080

Click **Create**.

**Install Jenkins**

To install latest version of Jenkins, add the repository key to the system and add the repository address to the sources list.

sudo wget -q -O - https://pkg.jenkins.io/debian/jenkins.io.key | sudo apt-key add -

sudo sh -c 'echo deb http://pkg.jenkins.io/debian-stable binary/ > /etc/apt/sources.list.d/jenkins.list'

Now you can update and install Jenkins.

sudo apt update

sudo apt install jenkins

**Starting Jenkins**

Once the installation is complete you can start Jenkins using the following command.

sudo systemctl start jenkins

You can also view the status of Jenkins using this command.

sudo systemctl status jenkins

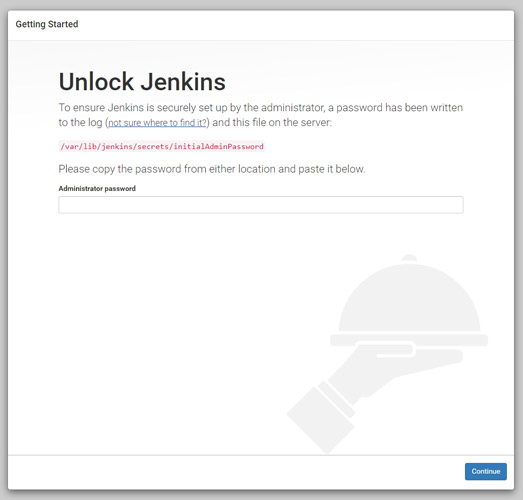
If Jenkins is started successfully you will get a response similar to this.

**Set Up Jenkins**

Once everything is done you can open your browser and enter your IP address followed by the Jenkins port 8080

The format will be like this http://instance\_external\_ip:8080

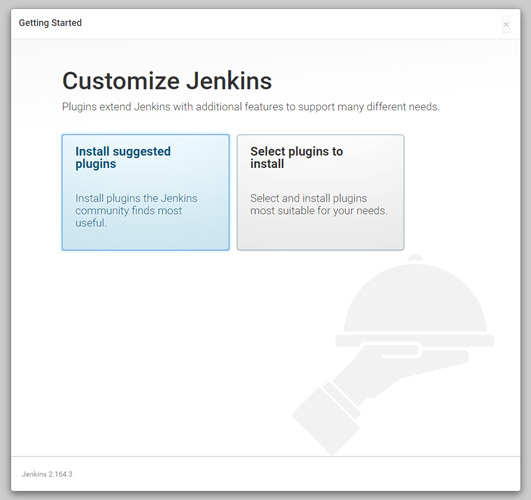
You will see the Unlock screen where you need to type the password to unlock Jenkins.



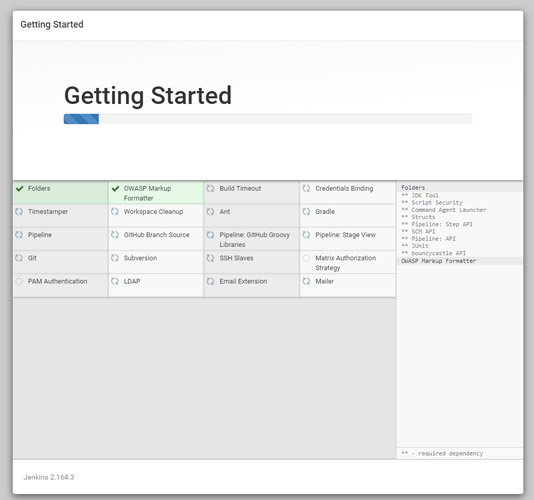
Execute the following command to get the password.

sudo cat /var/lib/jenkins/secrets/initialAdminPassword

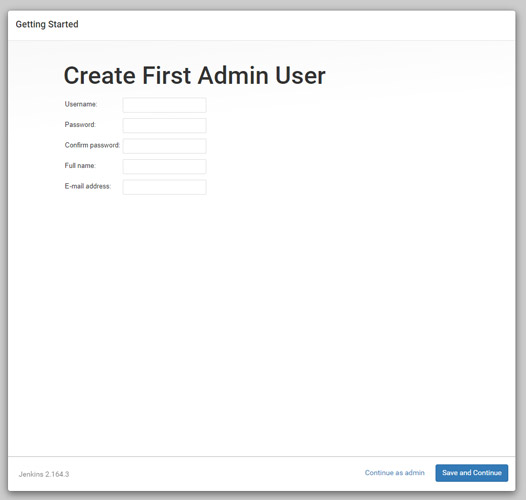
Copy the password and paste it in the **Administrator password** field to unlock and start the setup.



Click **Install suggested plugins** option to start the installation immediately.



Once the installation is complete you can create an admin user to login to the dashboard.



Finally, you will see the Instance Configuration, you can use your domain name or IP address.

Click **Save and Finish**.

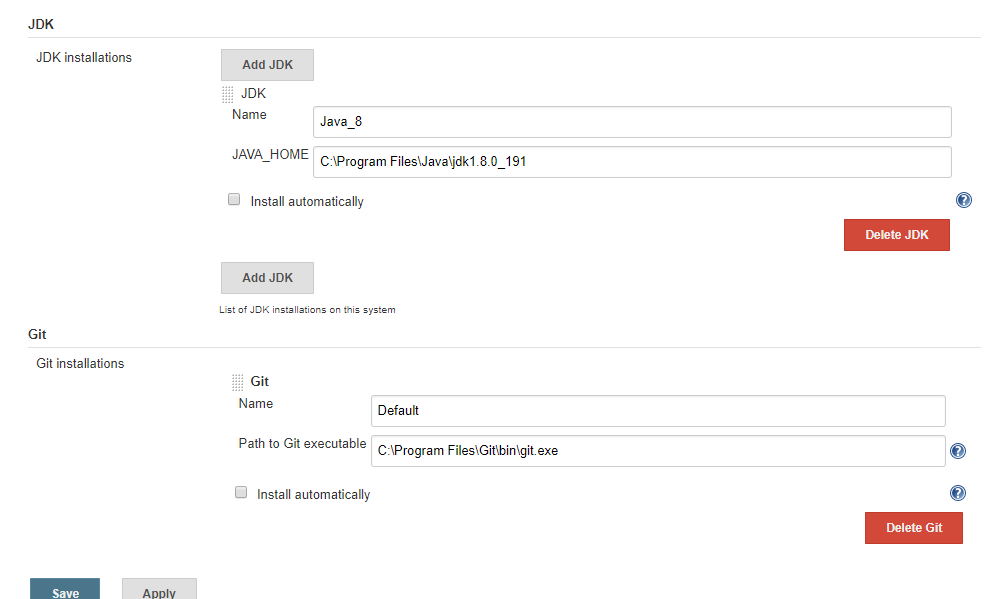
Once everything is complete click **Start using Jenkin**s to visit the main Jenkins dashboard.

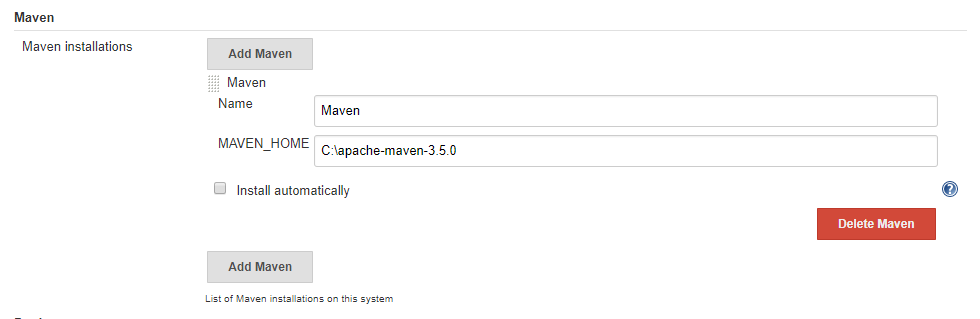
**Step 2: Integration of Github and Maven with Jenkins:**

**1.Configure Jenkins**

* Insure that **GitHub Plugin** is installed under- **Manage Jenkins** > **Manage Plugins** > **Installed** search for git. If not installed move to **Available** Tab and search for git and install it.

1. Configure *Java*, *GitHub* and *Maven* for Jenkins
2. Navigate to **Manage Jenkins** > **Global Tool Configuration**> Under **JDK** section provide *Name* and path to *JAVA\_HOME*, in same way for Git provide *Git Name* and path to Git executable, same in case of **Maven** provide *Name* and *MAVEN\_HOME* as in below images





**2.Create Job**

1. Create a new Job by clicking **New Item**

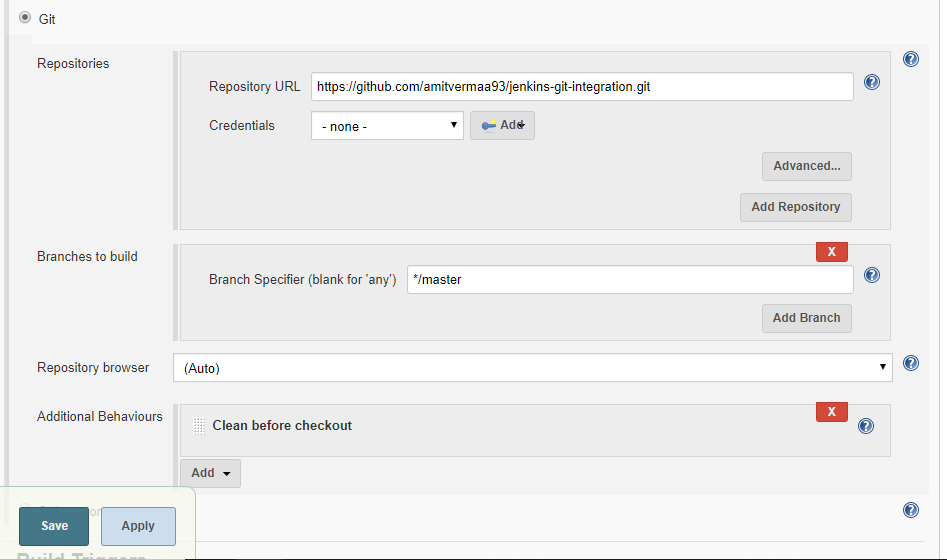
2. Enter your Job Name **‘Jenkins-GitHub’**and select ***Freestyle Project*** then click **OK**, You will be navigated to configure the Job.

3. In job configuration under **General** Section tick **Github** project and provide your *project url*from Github- **https://github.com/AbithaValli/quiz.git**this will provide you link to GitHub from Job dashboard and it is optional

4. Under **Source Code Management** section click on **Git** radio button and provide Repository URL- [**https://github.com/AbithaValli/quiz.git**](https://github.com/AbithaValli/quiz.git)

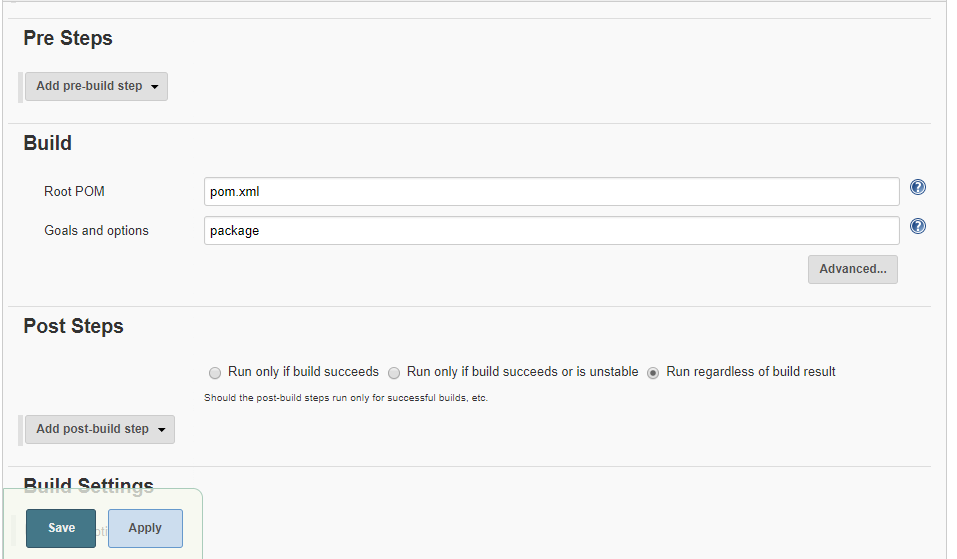
5. Select *branch* to build if master- ***‘\*/main’*** if development- ***‘\*/development’***

6. You can add *behavior* by selecting value from drop down what to perform in my case I want to clean space before code checkout so I have selected ***Clean before chekout*** (you can configure according to your need this is optional)



7. Navigate to **Build Trigger**- If you want to build your project on *specific time interval* you can configure it under **Build Trigger** > tick **Poll SCM** and in text field ***\* \* \* \* \**** (five star separated by space for every minute, see bottom of this article for convention). If you will not set up Poll you will have to manually build the Job.

8. Navigate to **Build** Section > provide path to***pom.xml***(for maven project) and *Goals and Options* **(**you can provide relative or absolute path to pom.xml, ***Relative*** : from project checkout directory as in below image, **Absolute** : from base directory like ***https://github.com/AbithaValli/quiz.git/pom.xml***and for goals you can define as you have set in your ***pom.xml***like **clean**, **package**, etc**)**



Navigate to that project. You will have to click on **Build Now** if you have not configured *Poll SCM* else it will automatically trigger every minute (as we have set \* \* \* \* \*). Your project will build successfully.

By clicking on *blue circle* you will be able to see logs.

By Clicking **#1** (#<Build Number>) you will be navigation to below window showing various project related information.

**3.Creating a pipeline job in Jenkins:**

## **Create your Pipeline project in Jenkins**

1. Go back to Jenkins, log in again if necessary and click **create new jobs** under **Welcome to Jenkins!**  
   **Note:** If you don’t see this, click **New Item** at the top left.
2. In the **Enter an item name** field, specify the name for your new Pipeline project (e.g. simple-java-maven-app).
3. Scroll down and click **Pipeline**, then click **OK** at the end of the page.
4. ( Optional ) On the next page, specify a brief description for your Pipeline in the **Description** field (e.g. An entry-level Pipeline demonstrating how to use Jenkins to build a simple Java application with Maven.)
5. Click the **Pipeline** tab at the top of the page to scroll down to the **Pipeline** section.
6. From the **Definition** field, choose the **Pipeline script from SCM** option. This option instructs Jenkins to obtain your Pipeline from Source Control Management (SCM), which will be your locally cloned Git repository.
7. From the **SCM** field, choose **Git**.
8. In the **Repository URL** field, specify the directory path of your locally cloned repository, which is from your user account/home directory on your host machine, mapped to the /home directory of the Jenkins container - i.e. - /home/Documents/GitHub/simple-java-maven-app
9. Click **Save** to save your new Pipeline project. You’re now ready to begin creating your Jenkinsfile, which you’ll be checking into your locally cloned Git repository.

# **4.Create your initial Pipeline as a Jenkinsfile**

You’re now ready to create your Pipeline that will automate building your Java application with Maven in Jenkins. Your Pipeline will be created as a Jenkinsfile, which will be committed to your locally cloned Git repository (simple-java-maven-app).

This is the foundation of “Pipeline-as-Code”, which treats the continuous delivery pipeline as a part of the application to be versioned and reviewed like any other code.

**Task 1:**

Create an initial Pipeline to download a Maven Docker image and run it as a Docker container (which will build your simple Java application). Also add a “Build” stage to the Pipeline that begins orchestrating this whole process.

Using any text editor , create and save new text file with the name Jenkinsfile at the root of your local simple-java-maven-app Git repository.

Copy the following Declarative Pipeline code and paste it into your empty Jenkinsfile:

|  |
| --- |
| pipeline { |
|  | agent any |
|  | environment { |
|  | PROJECT\_ID = 'serious-sylph-284014' |
|  | CLUSTER\_NAME = 'helloworld-gke' |
|  | LOCATION = 'us-west1-a' |
|  | registry = "ishu1108/docker-jenkins" |
|  | registryCredential = 'dockerhub' |
|  | dockerImage = '' |
|  | } |
|  |  |
|  | tools { |
|  | maven 'Maven-3.6.3' |
|  | jdk 'jdk8' |
|  | } |
|  |  |
|  |  |
|  |  |
|  |  |
|  | stages { |
|  | stage ('Initialize') { |
|  | steps { |
|  | sh ''' |
|  | echo "PATH = ${PATH}" |
|  | echo "M2\_HOME = ${M2\_HOME}" |
|  | ''' |
|  | } |
|  | } |
|  |  |
|  | stage ('Build-maven') { |
|  | steps { |
|  | echo 'This is a minimal pipeline.' |
|  | } |
|  | } |
|  |  |
|  |  |
|  |  |
|  | stage('Build') { |
|  | steps { |
|  | echo 'Building...' |
|  | echo "Running ${env.BUILD\_ID} ${env.BUILD\_DISPLAY\_NAME} on ${env.NODE\_NAME} and JOB ${env.JOB\_NAME}" |
|  | } |
|  | } |
|  | stage('Building image') { |
|  | steps{ |
|  | script { |
|  | dockerImage = docker.build registry |
|  | } |
|  | } |
|  | } |
|  |  |
|  | stage('Upload Docker Image to GCR'){ |
|  | steps{ |
|  | sh 'docker tag ishu1108/docker-jenkins gcr.io/serious-sylph-284014/certimage' |
|  | sh 'docker push gcr.io/serious-sylph-284014/certimage' |
|  | } |
|  | } |
|  |  |
|  | stage('Deploy to Kubernetes'){ |
|  | steps{ |
|  | sh 'gcloud container clusters get-credentials helloworld-gke \ |
|  | --zone us-west1-a \ |
|  | --project serious-sylph-284014' |
|  | sh 'kubectl apply -f deployment.yaml' |
|  | sh 'kubectl apply -f service.yaml' |
|  | } |
|  | } |
|  | } |
|  | } |
|  |  |

**Step 3: Creating CLUSTER in KUBERNETES:**

A GKE cluster is a managed set of Compute Engine virtual machines that operate as a single GKE cluster. Depending on the [mode of operation](https://cloud.google.com/kubernetes-engine/docs/concepts/types-of-clusters#modes) that you choose to use in GKE, when you create a cluster, you specify a default [zone or region](https://cloud.google.com/compute/docs/regions-zones#identifying_a_region_or_zone). If you use the Standard mode, your cluster is zonal (for this tutorial), so set your default compute [zone](https://cloud.google.com/compute/docs/regions-zones#available). If you use the Autopilot mode, your cluster is regional, so set your default compute [region](https://cloud.google.com/compute/docs/regions-zones#available).

gcloud container clusters create helloworld-gke \  
    --num-nodes 1 \  
    --zone *your-gcp-zone*

Verify that you have access to the cluster. The following command lists the nodes in your container cluster which are up and running and indicates that you have access to the cluster.

kubectl get nodes

**Step 4: Connecting Google Kubernetes Engine to a Cloud SQL instance:**

#### **Creating a Secret object**

1. You create the Secret objects by using the

[*kubectl create secret*](https://cloud.google.com/kubernetes-engine/docs/concepts/secret#creating_a_secret)

To create a database credentials Secret:

*kubectl create secret generic <YOUR-DB-SECRET> \  
  --from-literal=username=<YOUR-DATABASE-USER> \  
  --from-literal=password=<YOUR-DATABASE-PASSWORD> \  
  --from-literal=database=<YOUR-DATABASE-NAME>*

1. Once created, you can view the objects in the **Configuration** section of the Google Kubernetes Engine page in the [Cloud Console](https://console.cloud.google.com/kubernetes).

## **Connecting using the Cloud SQL Auth proxy**

When you connect using the Cloud SQL Auth proxy, the Cloud SQL Auth proxy is added to your pod using the sidecar container pattern. The Cloud SQL Auth proxy container is in the same pod as your application, which enables the application to connect to the Cloud SQL Auth proxy using localhost, increasing security and performance.

For connecting using the Cloud SQL Auth proxy you need the following:

1. The instance connection name of your Cloud SQL instance.

The instance connection name is available in the **Cloud SQL Instance details** page of the Cloud Console or from the gcloud sql instances describe **INSTANCE\_ID** command.

1. The location of the key file associated with a service account with the proper privileges for your Cloud SQL instance.
2. The Cloud SQL Admin API is enabled.

### **Providing the service account to the Cloud SQL Auth proxy**

The first step to running the Cloud SQL Auth proxy in Google Kubernetes Engine is creating a Google Service Account (GSA) to represent your application. It is recommended that you create a service account unique to each application, instead of using the same service account everywhere. This model is more secure since it allows you to limit permissions on a per-application basis.

The service account for your application needs to meet the following criteria:

* Belong to a project with the Cloud SQL Admin API enabled
* Has been granted the Cloud SQL Client IAM role (or equivalent) for the project containing the instance you want to connect to
* If connecting using private IP, you must use a VPC-native GKE cluster, in the same VPC as your Cloud SQL instance

You need to configure GKE to provide the service account to the Cloud SQL Auth proxy. There are two recommended ways to do this: [workload identity](https://cloud.google.com/sql/docs/mysql/connect-kubernetes-engine#workload-identity) or a [service account key file](https://cloud.google.com/sql/docs/mysql/connect-kubernetes-engine#service-account-key-file).

#### **Service account key file**

Alternatively, if you can't use Workload Identity, the recommended pattern is to mount a service account key file into the Cloud SQL Auth proxy pod and use the -credential\_file flag.

1. Create a credential file for your service account key:

*gcloud iam service-accounts keys create ~/key.json \  
  --iam-account <YOUR-SA-NAME>@project-id.iam.gserviceaccount.com*

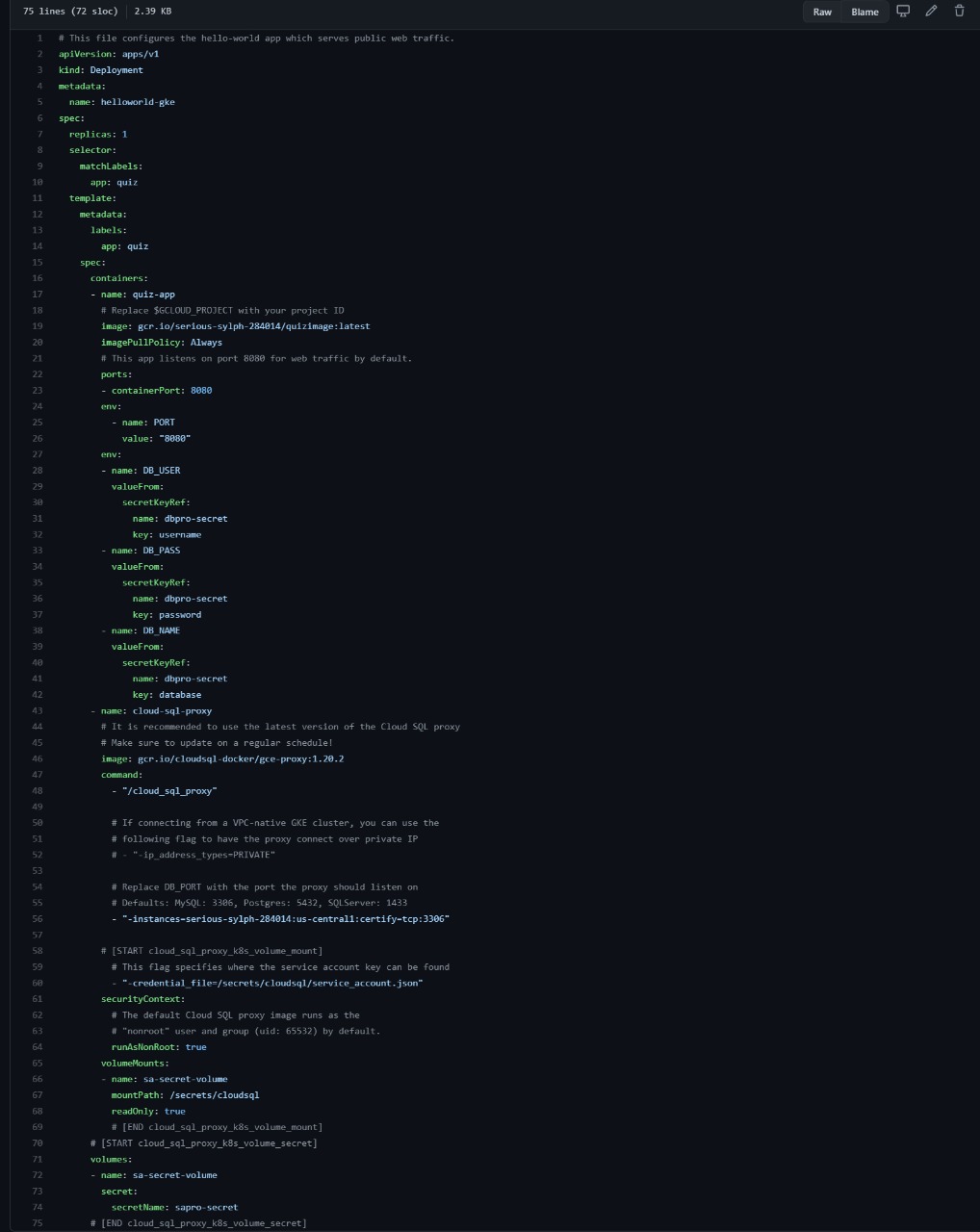
1. Turn your service account key into a k8s [Secret](https://kubernetes.io/docs/concepts/configuration/secret/):

*kubectl create secret generic <YOUR-SA-SECRET> \  
--from-file=service\_account.json=~/key.json*

### **Step 5: Deploy an app**

The app has a frontend server that handles the web requests. You define the cluster resources needed to run the frontend in a new file called deployment.yaml. These resources are described as a Deployment. You use Deployments to create and update a [ReplicaSet](https://kubernetes.io/docs/concepts/workloads/controllers/replicaset/) and its associated Pods.

1. Create the deployment.yaml file in the same directory as your other files and copy the following content, replacing **$GCLOUD\_PROJECT** with your Google Cloud project ID:



1. Deploy the resource to the cluster:

*kubectl apply -f deployment.yaml*

1. Track the status of the Deployment:

*kubectl get deployments*

The Deployment is complete, when all of the AVAILABLE deployments are READY.

*NAME READY UP-TO-DATE AVAILABLE AGE*

*hello-deployment 1/1 1 1 20s*

If the Deployment has a mistake, run kubectl apply -f deployment.yaml again to update the Deployment with any changes.

1. After the Deployment is complete, you can see the Pods that the Deployment created:

*kubectl get pods*

### **Step 6 : Deploy a Service**

[Services](https://kubernetes.io/docs/concepts/services-networking/service/) provide a single point of access to a set of Pods. While it's possible to access a single Pod, Pods are ephemeral and can only be accessed reliably by using a Service address. In your Hello World app, the "hello" Service defines a [load balancer](https://kubernetes.io/docs/tasks/access-application-cluster/create-external-load-balancer/) to access the hello-app Pods from a single IP address. This Service is defined in the service.yaml file.

*# The hello service provides a load-balancing proxy over the hello-app  
# pods. By specifying the type as a 'LoadBalancer', Kubernetes Engine will  
# create an external HTTP load balancer.  
apiVersion: v1  
kind: Service  
metadata:  
  name: hello  
spec:  
  type: LoadBalancer  
  selector:  
    app: hello  
  ports:  
  - port: 80  
    targetPort: 8080*

1. The Pods are defined separately from the Service that uses the Pods. Kubernetes uses [labels](https://kubernetes.io/docs/concepts/overview/working-with-objects/labels/) to select the Pods that a Service addresses. With labels, you can have a Service that addresses Pods from different replica sets and have multiple Services that point to an individual Pod.
2. Create the Hello World Service:

*kubectl apply -f service.yaml*

1. Get the Service's external IP address:

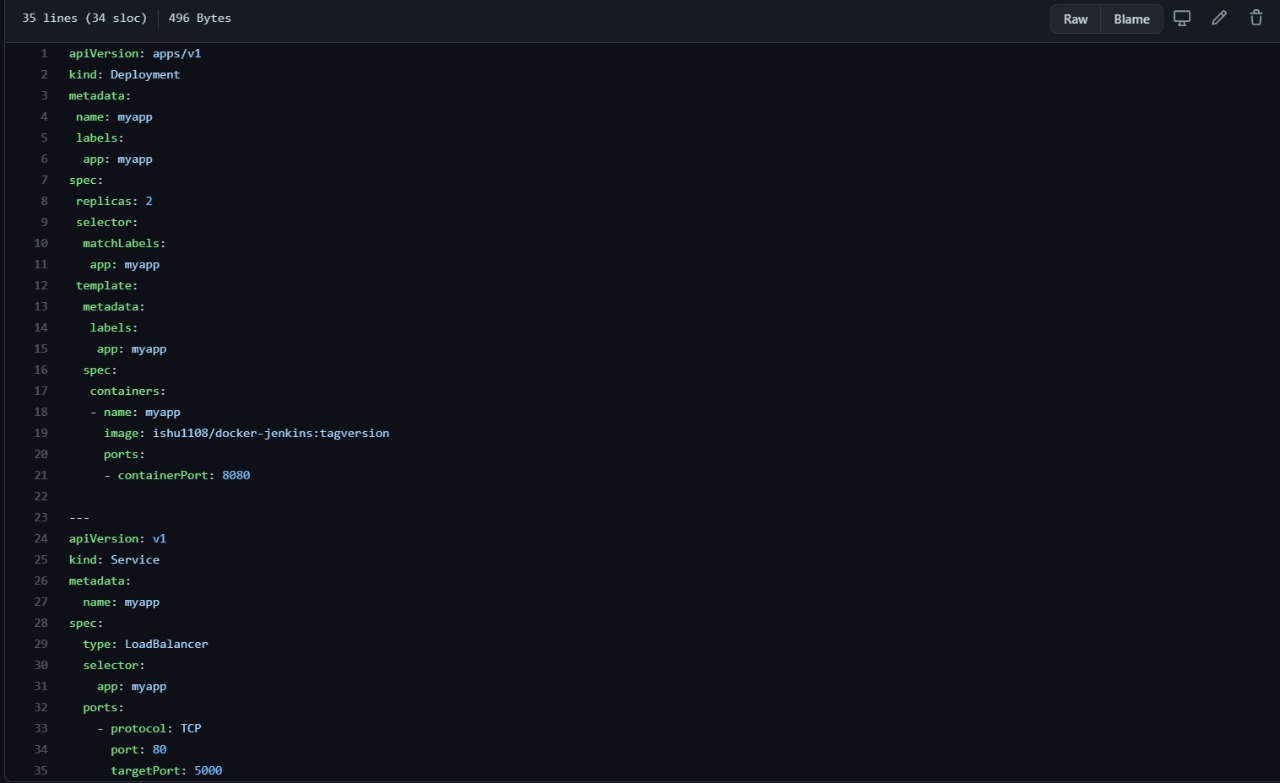
*kubectl get services*

**DEPLOYING ANOTHER APPLICATION IN THE SAME CLUSTER:**

### Follow the same steps till STEP 4: **Deploy an app**

The app has a frontend server that handles the web requests. You define the cluster resources needed to run the frontend in a new file called deployment.yaml. These resources are described as a Deployment. You use Deployments to create and update a [ReplicaSet](https://kubernetes.io/docs/concepts/workloads/controllers/replicaset/) and its associated Pods.

1. Create the deployment.yaml file in the same directory as your other files and copy the following content, replacing **$GCLOUD\_PROJECT** with your Google Cloud project ID:



Deploy the resource to the cluster:

*kubectl apply -f deployment.yaml*

Track the status of the Deployment:

*kubectl get deployments*

The Deployment is complete, when all of the AVAILABLE deployments are READY.

*NAME READY UP-TO-DATE AVAILABLE AGE*

*hello-deployment 1/1 1 1 20s*

If the Deployment has a mistake, run kubectl apply -f deployment.yaml again to update the Deployment with any changes.

After the Deployment is complete, you can see the Pods that the Deployment created:

*kubectl get pods*

### **Deploy a Service**

[Services](https://kubernetes.io/docs/concepts/services-networking/service/) provide a single point of access to a set of Pods. While it's possible to access a single Pod, Pods are ephemeral and can only be accessed reliably by using a Service address. In your Hello World app, the "hello" Service defines a [load balancer](https://kubernetes.io/docs/tasks/access-application-cluster/create-external-load-balancer/) to access the hello-app Pods from a single IP address. This Service is defined in the service.yaml file.

*# The hello service provides a load-balancing proxy over the hello-app  
# pods. By specifying the type as a 'LoadBalancer', Kubernetes Engine will  
# create an external HTTP load balancer.  
apiVersion: v1  
kind: Service  
metadata:  
  name: hello  
spec:  
  type: LoadBalancer  
  selector:  
    app: hello  
  ports:  
  - port: 80  
    targetPort: 8080*

The Pods are defined separately from the Service that uses the Pods. Kubernetes uses [labels](https://kubernetes.io/docs/concepts/overview/working-with-objects/labels/) to select the Pods that a Service addresses. With labels, you can have a Service that addresses Pods from different replica sets and have multiple Services that point to an individual Pod.

Create the Hello World Service:

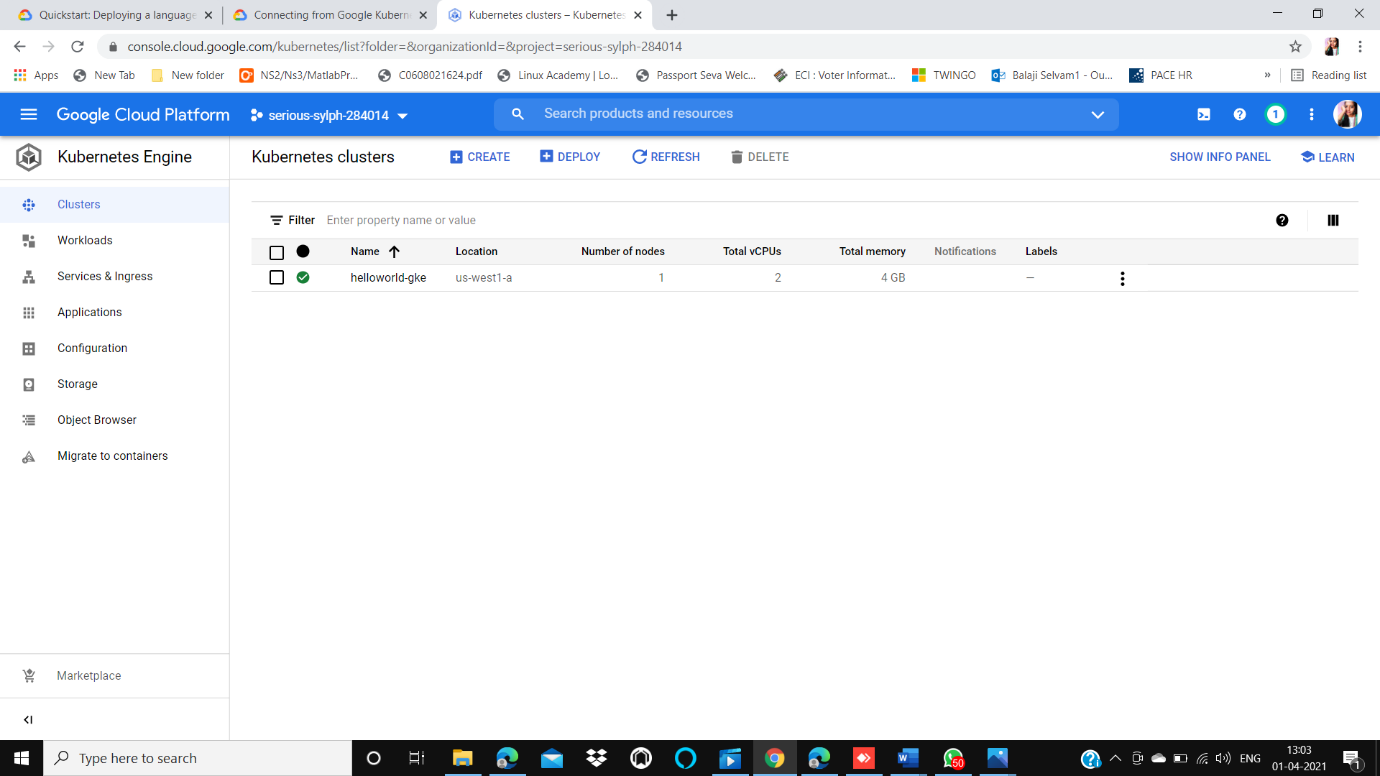
*kubectl apply -f service.yaml*

Get the Service's external IP address:

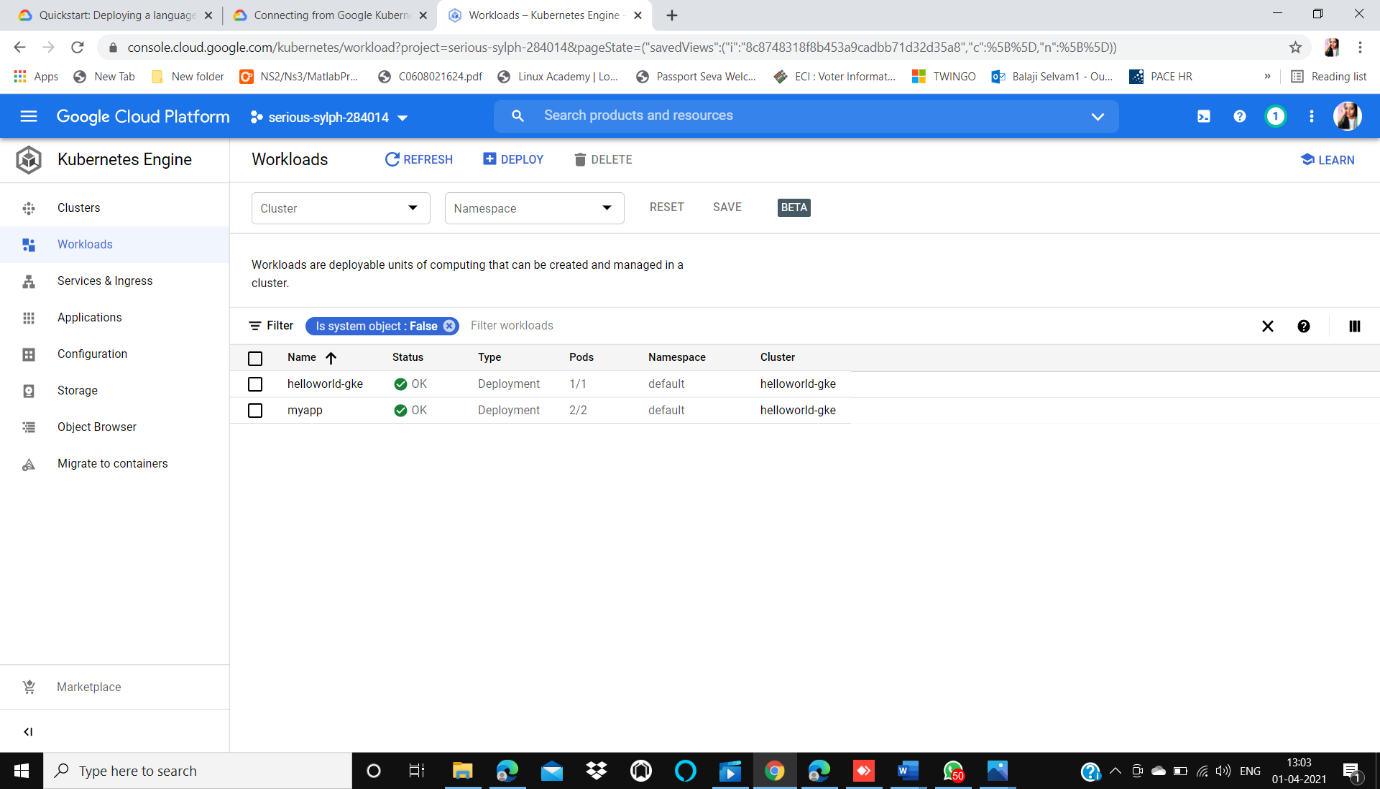
*kubectl get services*

**SCREENSHOTS:**

**CLUSTER**



**WORKLOADS:**



**SERVICES:**

