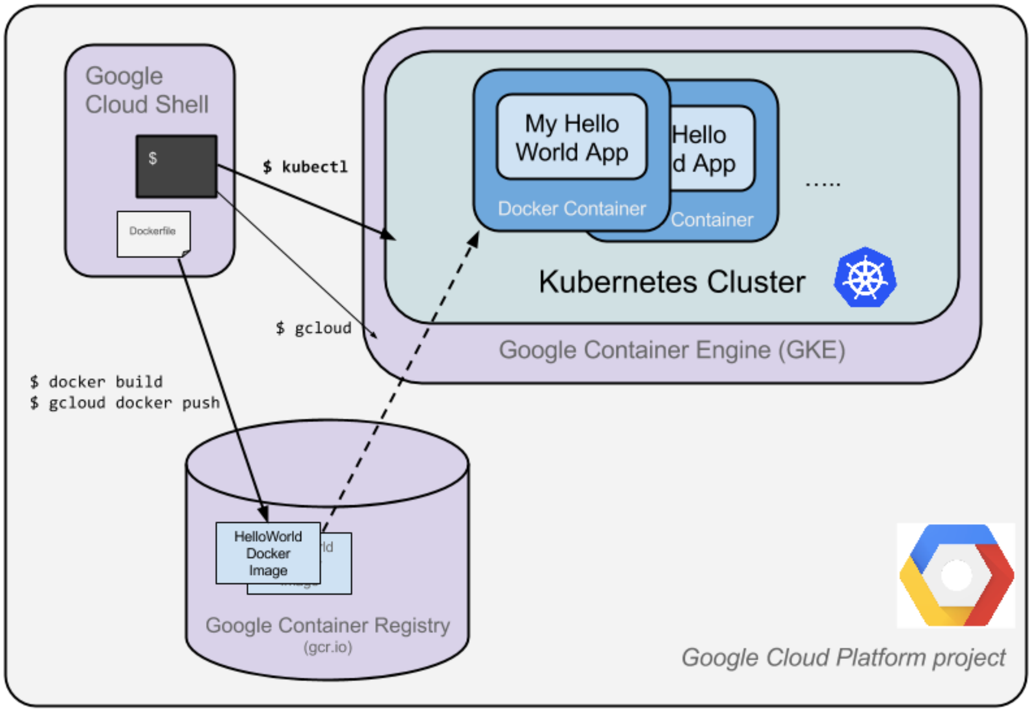
Hello Node Kubernetes

GSP005

Google Cloud Self-Paced Labs

Overview

The goal of this hands-on lab is for you to turn code that you have developed into a replicated application running on Kubernetes, which is running on Kubernetes Engine. For this lab the code will be a simple Hello World node.js app.
Here's a diagram of the various parts in play in this lab, to help you understand how the pieces fit together with one another. Use this as a reference as you progress through the lab; it should all make sense by the time you get to the end (but feel free to ignore this for now).



Kubernetes is an open source project (available on kubernetes.io) which can run on many different environments, from laptops to high-availability multi-node clusters; from public clouds to on-premise deployments; from virtual machines to bare metal.
For the purpose of this lab, using a managed environment such as Kubernetes Engine (a Google-hosted version of Kubernetes running on Compute Engine) will

allow you to focus more on experiencing Kubernetes rather than setting up the underlying infrastructure.

What you'll do

- Create a Node.js server.
- Create a Docker container image.
- Create a container cluster.
- Create a Kubernetes pod.
- Scale up your services.

Prerequisites

- Familiarity with standard Linux text editors such as `vim`, `emacs`, or `nano` will be helpful.
  We encourage students to type the commands themselves, to help encourage learning of the core concepts. Many labs will include a code block that contains the required commands. You can easily copy and paste the commands from the code block into the appropriate places during the lab.

Create your Node.js application

Using Cloud Shell, write a simple Node.js server that you'll deploy to Kubernetes Engine:

```
vi server.js
```

Start the editor:

```
i
```

Add this content to the file:

```
var http = require('http');
var handleRequest = function(request, response) {
  response.writeHead(200);
  response.end("Hello World!");
}
var www = http.createServer(handleRequest);
www.listen(8080);
```

Note: `vi` is used here, but `nano` and `emacs` are also available in Cloud Shell. You can also use the Web-editor feature of CloudShell as described here.

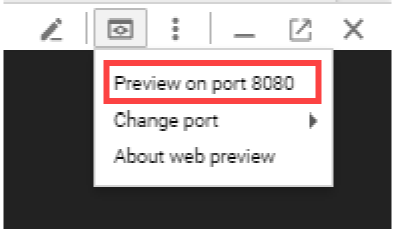
Save the `server.js` file: **Esc** then:

```
:wq
```

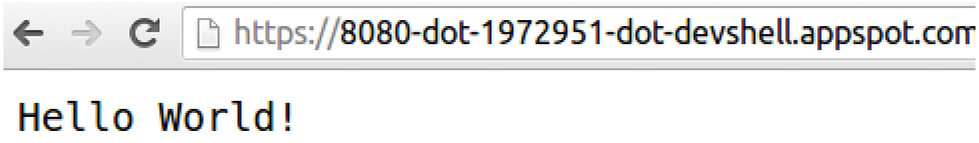
Since Cloud Shell has the `node` executable installed, run this command to start the node server (the command produces no output):

```
node server.js
```

Use the built-in Web preview feature of Cloud Shell to open a new browser tab and proxy a request to the instance you just started on port `8080`.



A new browser tab will open to display your results:



https://8080-dot-1972951-dot-devshell.appspot.com

```
Hello World!
```

Before continuing, return to Cloud Shell and type **Ctrl+c** to stop the running node server.

Next you will package this application in a Docker container.

Create a Docker container image

Next, create a `Dockerfile` that describes the image you want to build. Docker container images can extend from other existing images, so for this image, we'll extend from an existing Node image.

```
vi Dockerfile
```

Start the editor:

```
i
```

Add this content:

```
FROM node:6.9.2
EXPOSE 8080
COPY server.js .
CMD node server.js
```

This "recipe" for the Docker image will:

- Start from the `node` image found on the Docker hub.
- Expose port `8080`.
- Copy your `server.js` file to the image.
- Start the node server as we previously did manually.

Save this `Dockerfile` by pressing **Esc**, then type:

```
:wq
```

Build the image with the following, replacing `PROJECT_ID` with your GCP Project ID, found in the Console and the **Connection Details** section of the lab:

```
docker build -t gcr.io/PROJECT_ID/hello-node:v1 .
```

It'll take some time to download and extract everything, but you can see the progress bars as the image builds.

Once complete, test the image locally by running a Docker container as a daemon on port 8080 from your newly-created container image.

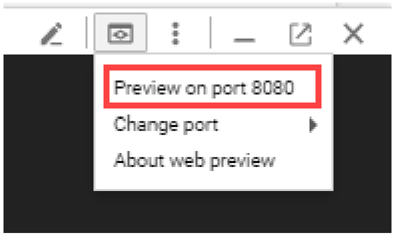
Run the following command replacing `PROJECT_ID` with your GCP Project ID, found in the Console and the **Connection Details** section of the lab:

```
docker run -d -p 8080:8080 gcr.io/PROJECT_ID/hello-node:v1
```

Your output should look something like this:

```
325301e6b2bffd1d0049c621866831316d653c0b25a496d04ce0ec6854cb7998
```

To see your results you can use the web preview feature of Cloud Shell:



Or use `curl` from your Cloud Shell prompt:

```
curl http://localhost:8080
```

This is the output you should see:

```
Hello World!
```

Note: Full documentation for the `docker run` command is found here.

Next, stop the running container.

Find your Docker container ID by running:

```
docker ps
```

Your output you should look like this:

```
CONTAINER ID         IMAGE                              COMMAND
2c66d0efcbd4         gcr.io/PROJECT_ID/hello-node:v1    "/bin/sh -c 'node
```

Stop the container by running the following, replacing the `[CONTAINER ID]` with the value provided from the previous step:

```
docker stop [CONTAINER ID]
```

Your console output should resemble the following (your container ID):

```
2c66d0efcbd4
```

Now that the image is working as intended, push it to the Google Container Registry, a private repository for your Docker images, accessible from your Google Cloud projects.

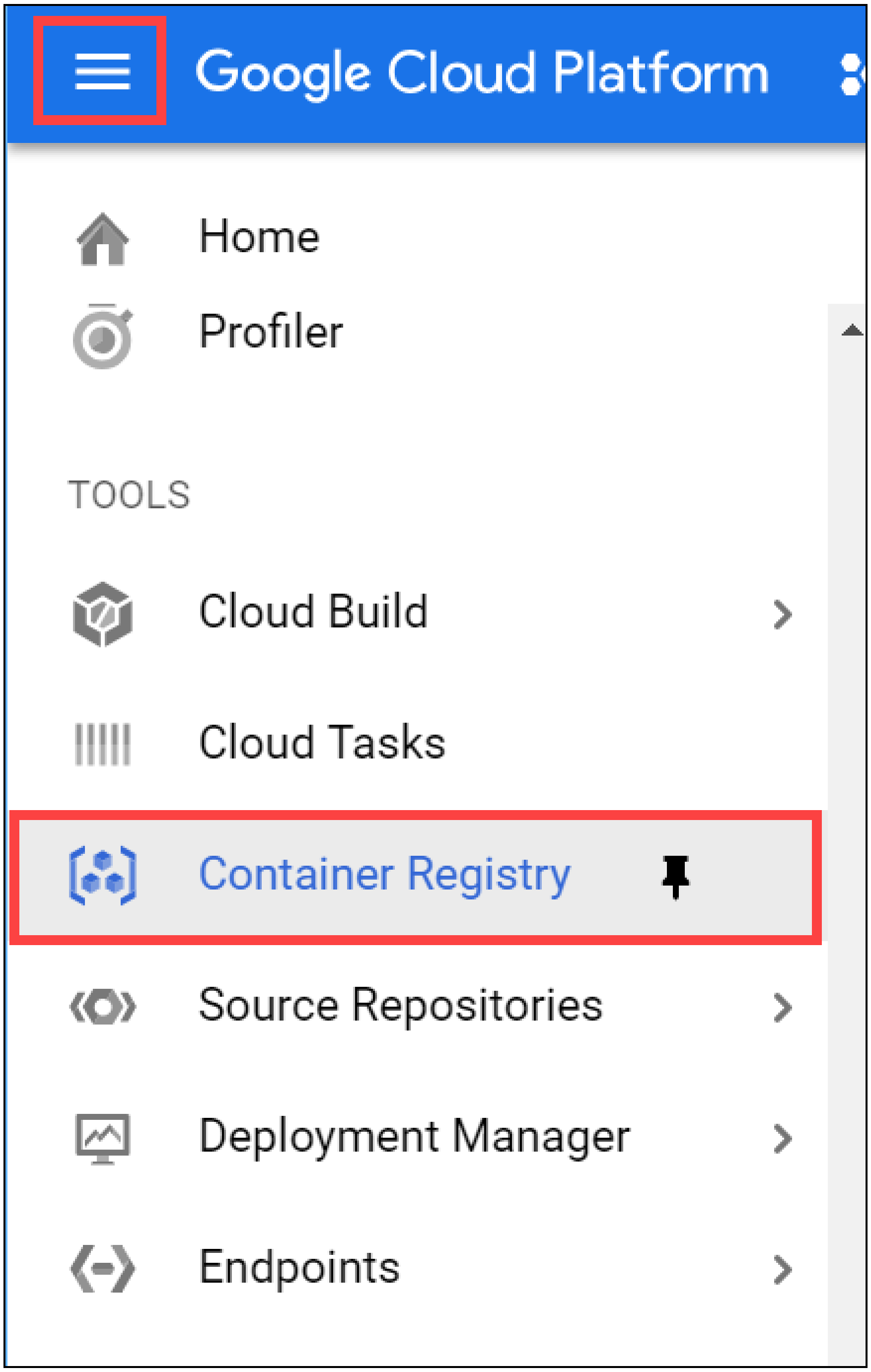
Run this command, replacing `PROJECT_ID` with your GCP Project ID, found in the Console or the **Connection Details** section of the lab.

```
gcloud auth configure-docker
docker push gcr.io/PROJECT_ID/hello-node:v1
```

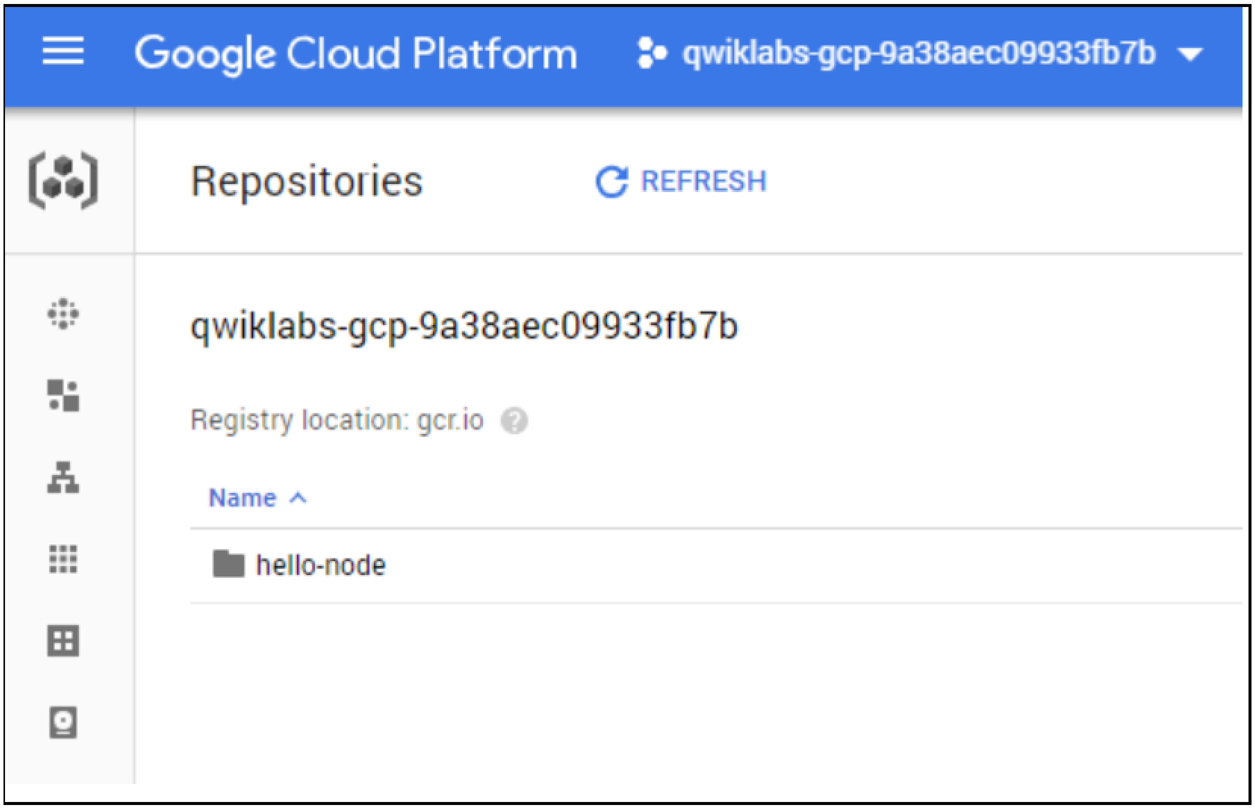
The initial push may take a few minutes to complete. You'll see the progress bars as it builds.

```
The push refers to a repository [gcr.io/qwiklabs-gcp-6h281a111f098/hello-
node]
ba6ca48af64e: Pushed
381c97ba7dc3: Pushed
604c78617f34: Pushed
fa18e5ffd316: Pushed
0a5e2b2ddeaa: Pushed
53c779688d06: Pushed
60a0858edcd5: Pushed
b6ca02dfe5e6: Pushed
v1: digest:
sha256:8a9349a355c8e06a48a1e8906652b9259bba6d594097f115060acca8e3e941a2 size:
2002
```

The container image will be listed in your Console. Select **Navigation menu** > **Container Registry**.



Now you have a project-wide Docker image available which Kubernetes can access and orchestrate.



qwiklabs-gcp-9a38aec09933fb7b

Registry location: gcr.io

Name ^
hello-node

Note: A generic domain is used for the registry (`gcr.io`). In your own environment you can be more specific about which zone and bucket to use. Details are documented here.

Create your cluster

Now you're ready to create your Kubernetes Engine cluster. A cluster consists of a Kubernetes master API server hosted by Google and a set of worker nodes. The worker nodes are Compute Engine virtual machines.

Make sure you have set your project using `gcloud` (replace `PROJECT_ID` with your GCP Project ID, found in the console and in the **Connection Details** section of the lab):

```
gcloud config set project PROJECT_ID
```

Create a cluster with two n1-standard-1 nodes (this will take a few minutes to complete):

```
gcloud container clusters create hello-world \
                --num-nodes 2 \
                --machine-type n1-standard-1 \
                --zone us-central1-a
```

You can safely ignore warnings that come up when the cluster builds.

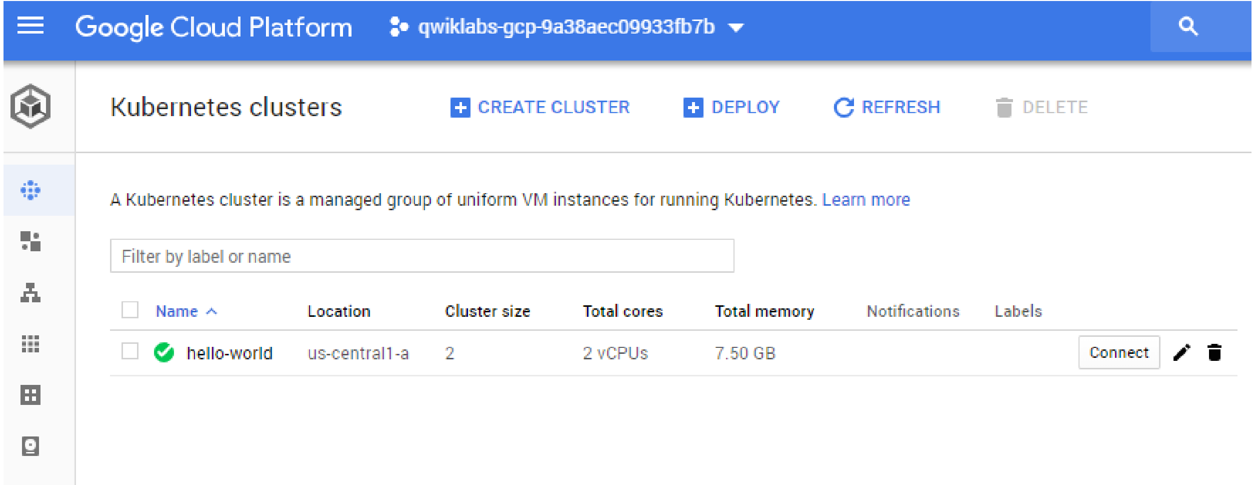
The console output should look like this:

```
Creating cluster hello-world...done.
Created [https://container.googleapis.com/v1/projects/PROJECT_ID/zones/us-
central1-a/clusters/hello-world].
kubeconfig entry generated for hello-world.
NAME         ZONE           MASTER_VERSION  MASTER_IP       MACHINE_TYPE
STATUS
hello-world  us-central1-a  1.5.7           146.148.46.124  n1-standard-1
RUNNING
```

Note: You can also create this cluster through the Console by opening the Navigation menu and selecting **Kubernetes Engine** > **Kubernetes clusters** > **Create cluster**.

Note: It is recommended to create the cluster in the same zone as the storage bucket used by the container registry (see previous step).

If you select **Navigation menu** > **Kubernetes Engine**, you'll see that you now you have a fully-functioning Kubernetes cluster powered by Kubernetes Engine:



A Kubernetes cluster is a managed group of uniform VM instances for running Kubernetes. Learn more

Name	Location	Cluster size	Total cores	Total memory	Notifications	Labels
hello-world	us-central1-a	2	2 vCPUs	7.50 GB		

It's time to deploy your own containerized application to the Kubernetes cluster! From now on you'll use the `kubectl` command line (already set up in your Cloud Shell environment).

Click **Check my progress** below to check your lab progress.

Create your cluster.

Check my progress

Create your pod

A Kubernetes **pod** is a group of containers tied together for administration and networking purposes. It can contain single or multiple containers. Here you'll use one container built with your Node.js image stored in your private container registry. It will serve content on port 8080.

Create a pod with the `kubectl run` command (replace `PROJECT_ID` with your GCP Project ID, found in the console and in the **Connection Details** section of the lab):

```
kubectl run hello-node \
    --image=gcr.io/PROJECT_ID/hello-node:v1 \
    --port=8080
```

(Output)

```
deployment "hello-node" created
```

As you can see, you've created a **deployment** object. Deployments are the recommended way to create and scale pods. Here, a new deployment manages a single pod replica running the `hello-node:v1` image.

To view the deployment, run:

```
kubectl get deployments
```

(Output)

```
NAME          DESIRED    CURRENT    UP-TO-DATE    AVAILABLE    AGE
hello-node    1          1          1             1            2m
```

To view the pod created by the deployment, run:

```
kubectl get pods
```

(Output)

```
NAME                           READY      STATUS     RESTARTS     AGE
hello-node-714049816-ztzrb     1/1        Running    0            6m
```

Now is a good time to go through some interesting `kubectl` commands. None of these will change the state of the cluster, full documentation is available here:

```
kubectl cluster-info
kubectl config view
```

And for troubleshooting :

```
kubectl get events
kubectl logs <pod-name>
```

You now need to make your pod accessible to the outside world.

Allow external traffic

By default, the pod is only accessible by its internal IP within the cluster. In order to make the `hello-node` container accessible from outside the Kubernetes virtual network, you have to expose the pod as a Kubernetes **service**.
From Cloud Shell you can expose the pod to the public internet with the `kubectl expose` command combined with the `--type="LoadBalancer"` flag. This flag is required for the creation of an externally accessible IP:

```
kubectl expose deployment hello-node --type="LoadBalancer"
```

(Output)

```
service "hello-node" exposed
```

The flag used in this command specifies that are using the load-balancer provided by the underlying infrastructure (in this case the Compute Engine load balancer). Note that you expose the deployment, and not the pod, directly. This will cause the resulting service to load balance traffic across all pods managed by the deployment (in this case only 1 pod, but you will add more replicas later).
The Kubernetes master creates the load balancer and related Compute Engine forwarding rules, target pools, and firewall rules to make the service fully accessible from outside of Google Cloud Platform.

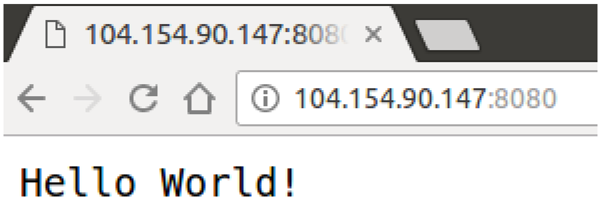
To find the publicly-accessible IP address of the service, request `kubectl` to list all the cluster services:

```
kubectl get services
```

This is the output you should see:

```
NAME          CLUSTER-IP     EXTERNAL-IP      PORT(S)    AGE
hello-node    10.3.250.149   104.154.90.147   8080/TCP   1m
kubernetes    10.3.240.1     <none>           443/TCP    5m
```

There are 2 IP addresses listed for your hello-node service, both serving port 8080. The `CLUSTER-IP` is the internal IP that is only visible inside your cloud virtual network; the `EXTERNAL-IP` is the external load-balanced IP.
Note: The `EXTERNAL-IP` may take several minutes to become available and visible. If the `EXTERNAL-IP` is missing, wait a few minutes and run the command again.
You should now be able to reach the service by pointing your browser to this address: `http://<EXTERNAL_IP>:8080`



```
Hello World!
```

At this point you've gained several features from moving to containers and Kubernetes - you do not need to specify on which host to run your workload and you also benefit from service monitoring and restart. Now see what else can be gained from your new Kubernetes infrastructure.

Scale up your service

One of the powerful features offered by Kubernetes is how easy it is to scale your application. Suppose you suddenly need more capacity. You can tell the replication controller to manage a new number of replicas for your pod:

```
kubectl scale deployment hello-node --replicas=4
```

(Output)

```
deployment "hello-node" scaled
```

You can request a description of the updated deployment:

```
kubectl get deployment
```

(Output)

```
NAME          DESIRED    CURRENT    UP-TO-DATE    AVAILABLE    AGE
hello-node    4          4          4             4            16m
```

You can also list the all pods:

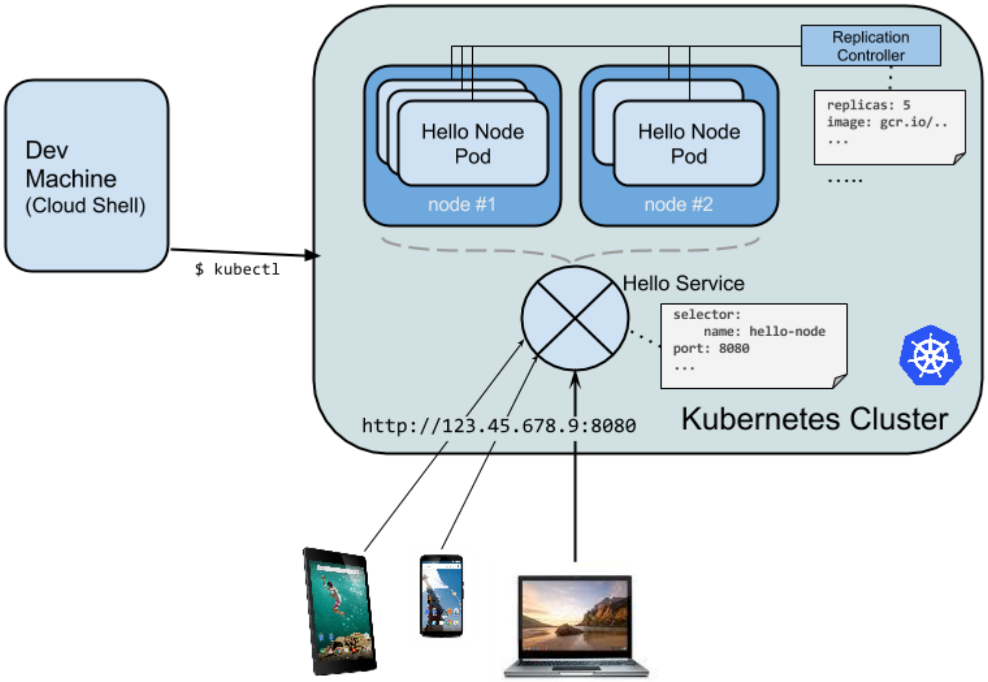
```
kubectl get pods
```

This is the output you should see:

```
NAME                           READY      STATUS     RESTARTS    AGE
hello-node-714049816-g4azy     1/1        Running    0           1m
hello-node-714049816-rk0u6     1/1        Running    0           1m
hello-node-714049816-sh812     1/1        Running    0           1m
hello-node-714049816-ztzrb     1/1        Running    0           16m
```

A **declarative approach** is being used here. Rather than starting or stopping new instances, you declare how many instances should be running at all times. Kubernetes reconciliation loops makes sure that reality matches what you requested and takes action if needed.

Here's a diagram summarizing the state of your Kubernetes cluster:



Roll out an upgrade to your service

At some point the application that you've deployed to production will require bug fixes or additional features. Kubernetes helps you deploy a new version to production without impacting your users.

First, modify the application by opening `server.js`:

```
vi server.js
i
```

Then update the response message:

```
response.end("Hello Kubernetes World!");
```

Save the `server.js` file by pressing **Esc** then:

```
:wq
```

Now you can build and publish a new container image to the registry with an incremented tag (`v2` in this case).

Run the following commands, replacing `PROJECT_ID` with your lab project ID:

```
docker build -t gcr.io/PROJECT_ID/hello-node:v2 .
docker push gcr.io/PROJECT_ID/hello-node:v2
```

Note: Building and pushing this updated image should be quicker since caching is being taken advantage of.

Kubernetes will smoothly update your replication controller to the new version of the application. In order to change the image label for your running container, you will edit the existing `hello-node deployment` and change the image from `gcr.io/PROJECT_ID/hello-node:v1` to `gcr.io/PROJECT_ID/hello-node:v2`.

To do this, use the `kubectl edit` command. It opens a text editor displaying the full deployment yaml configuration. It isn't necessary to understand the full yaml config right now, just understand that by updating the `spec.template.spec.containers.image` field in the config you are telling the deployment to update the pods with the new image.

```
kubectl edit deployment hello-node
```

Look for `Spec` > `containers` > `image` and change the version number to v2:

```
# Please edit the object below. Lines beginning with a '#' will be ignored,
# and an empty file will abort the edit. If an error occurs while saving this
file will be
# reopened with the relevant failures.
#
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  annotations:
    deployment.kubernetes.io/revision: "1"
  creationTimestamp: 2016-03-24T17:55:28Z
  generation: 3
  labels:
    run: hello-node
  name: hello-node
  namespace: default
  resourceVersion: "151017"
  selfLink: /apis/extensions/v1beta1/namespaces/default/deployments/hello-
node
  uid: 981fe302-f1e9-11e5-9a78-42010af00005
spec:
  replicas: 4
  selector:
    matchLabels:
      run: hello-node
  strategy:
    rollingUpdate:
      maxSurge: 1
      maxUnavailable: 1
    type: RollingUpdate
  template:
    metadata:
      creationTimestamp: null
      labels:
        run: hello-node
    spec:
      containers:
      - image: gcr.io/PROJECT_ID/hello-node:v1 ## Update this line ##
        imagePullPolicy: IfNotPresent
        name: hello-node
        ports:
        - containerPort: 8080
          protocol: TCP
        resources: {}
        terminationMessagePath: /dev/termination-log
      dnsPolicy: ClusterFirst
      restartPolicy: Always
      securityContext: {}
      terminationGracePeriodSeconds: 30
```

After making the change, save and close this file: Press **Esc**, then:

```
:wq
```

This is the output you should see:

```
deployment "hello-node" edited
```

Run the following to update the deployment with the new image:

```
kubectl get deployments
```

New pods will be created with the new image and the old pods will be deleted.

This is the output you should see:

```
NAME         DESIRED   CURRENT   UP-TO-DATE   AVAILABLE   AGE
hello-node   4         4         4            4           1h
```

While this is happening, the users of your services shouldn't see any interruption. After a little while they'll start accessing the new version of your application. You can find more details on rolling updates in this documentation.

Hopefully with these deployment, scaling, and updated features, once you've set up your Kubernetes Engine cluster, you'll agree that Kubernetes will help you focus on the application rather than the infrastructure.

Kubernetes graphical dashboard (optional)

A graphical web user interface (dashboard) has been introduced in recent versions of Kubernetes. The dashboard allows you to get started quickly and enables some of the functionality found in the CLI as a more approachable and discoverable way of interacting with the system.

To get started, run the following command to grant cluster level permissions:

```
kubectl create clusterrolebinding cluster-admin-binding --
clusterrole=cluster-admin --user=$(gcloud config get-value account)
```

With the appropriate permissions set, run the following command to create a new dashboard service:

```
kubectl apply -f
https://raw.githubusercontent.com/kubernetes/dashboard/v1.10.1/src/deploy/rec
ommended/kubernetes-dashboard.yaml
```

You should receive a similar output:

```
secret "kubernetes-dashboard-certs" created
serviceaccount "kubernetes-dashboard" created
role.rbac.authorization.k8s.io "kubernetes-dashboard-minimal" created
rolebinding.rbac.authorization.k8s.io "kubernetes-dashboard-minimal" created
deployment.apps "kubernetes-dashboard" created
service "kubernetes-dashboard" created
```

Now run the following command to edit the `yaml` representation of the dashboard service:

```
kubectl -n kube-system edit service kubernetes-dashboard
```

Press `i` to enter the editing mode.
Change `type: ClusterIP` to `type: NodePort`.
After making the change, save and close this file. Press **Esc**, then:

```
:wq
```

To log in to the Kubernetes dashboard you must authenticate using a token. Use a token allocated to a service account, such as the `namespace-controller`.
To get the token value, run the following command:

```
kubectl -n kube-system describe $(kubectl -n kube-system \
get secret -n kube-system -o name | grep namespace) | grep token:
```

You should receive a similar Output:

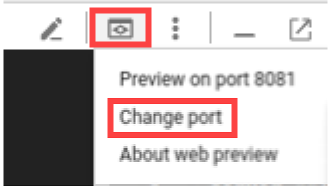
```
token:
eyJhbGciOiJSUzI1NiIsInR5cCI6IkpXVCJ9.eyJpc3MiOiJrdWJlcm5ldGVzL3NlcnZpY2VhY2Nv
dW50Iiwia3ViZXJuZXRlcy5pby9zZXJ2aWNlYWNjb3VudC9uYW1lc3BhY2UiOiJrdWJlLXN5c3Rlb
SIsImt1YmVybmV0ZXMuaW8vc2VydmljZWFjY291bnQvc2VjcmV0Lm5hbWUiOiJuYW1lc3BhY2UtY2
9udHJvbGxlci10b2tlbi1kOTZyNCIsImt1YmVybmV0ZXMuaW8vc2VydmljZWFjY291bnQvc2Vydml
```

```
jZS1hY2NvdW50Lm5hbWUiOiJuYW1lc3BhY2UtY29udHJvbGxlciIsImt1YmVybmV0ZXMuaW8vc2Vy
dmljZWFjY291bnQvc2VydmljZS1hY2NvdW50LnVpZCI6ImU2ZmFkNGQ5LTJjNjYtMTFlOC05NDFiL
TQyMDEwYTgwMDFlYiIsInN1YiI6InN5c3RlbTpzZXJ2aWNlYWNjb3VudDprdWJlLXN5c3RlbTpuYW
1lc3BhY2UtY29udHJvbGxlciJ9.AY3Fp-T_4wxTzvo4kiWi4zxojVTSr1Wy7BL_-
HmIRlWTRAUmy_1RAJS19zn4BbSkxlV13Y9Bv3NoVcG01jKd4QoM172OXo2TqSU5v2B62i3-
_CDZtf3CVgQIp9jiuxACcR5zg3w-
4ewGfH4C3ospoKCuayyRaADLq0ThWLGaTQv9e7UjSfWAPir3XPXQut3mMRYrSiHcFNiEGeztSfF3c
yhuvL2I5Lfh20yYuqW5j-w72BLnlqQGPuhJXJgH1_35XUCU8WtnkEK-qYX40ajDWJYa1s9_R-
MWzF6Zwji2Gh5txOvxG3lZuIq9GSAOBp85617wB3eCGio6Nu3L9TwWXA
```

Copy the token and save it to use later to get into the Kubernetes dashboard. Run the following command to open a connection:

```
kubectl proxy --port 8081
```

Then use the Cloud Shell Web preview feature to change ports to **8081**:



This should send you to the API endpoint.

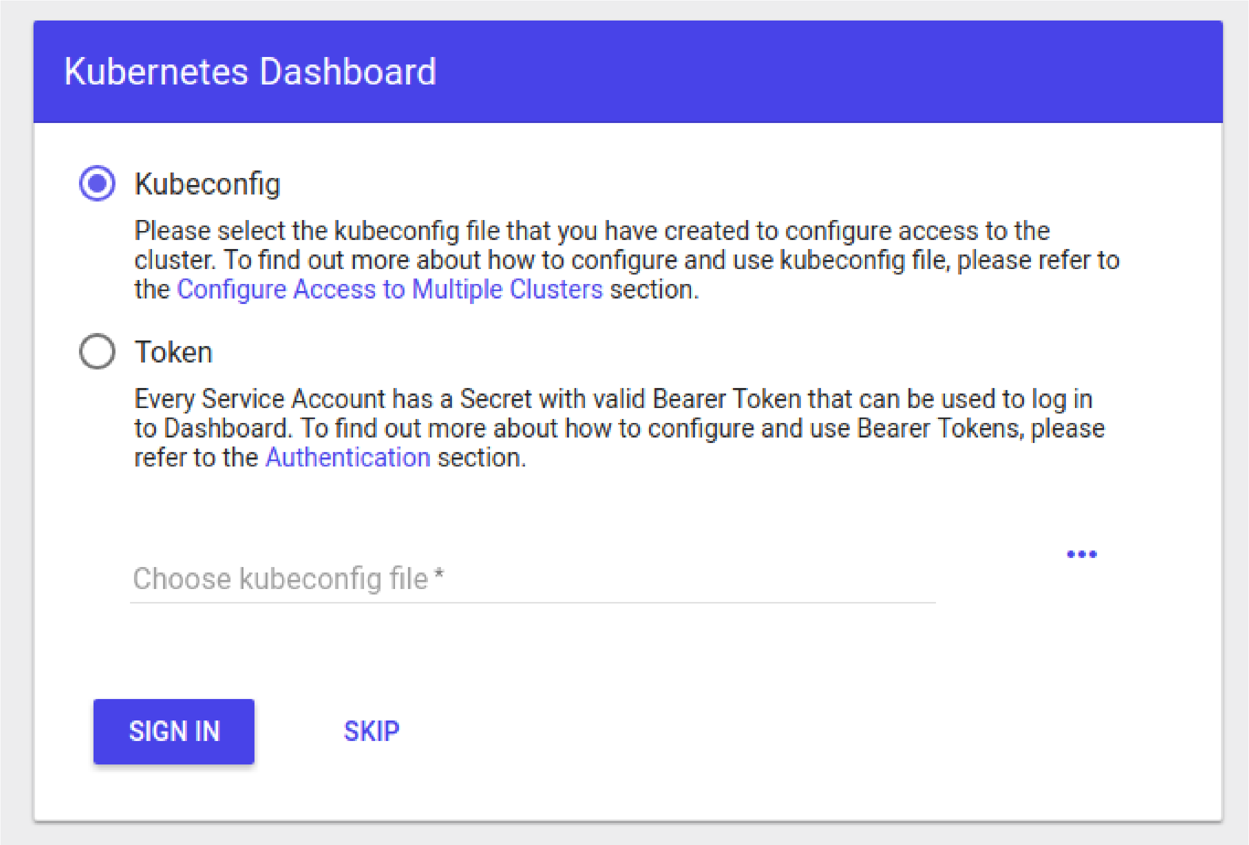
To get to the dashboard, remove `/?authuser=0` and append the URL with the following:

```
/api/v1/namespaces/kube-system/services/https:kubernetes-
dashboard:/proxy/#!/overview?namespace=default
```

Your final URL should resemble the following:

```
https://8081-dot-5177448-dot-devshell.appspot.com/api/v1/namespaces/kube-
system/services/https:kubernetes-
dashboard:/proxy/#!/overview?namespace=default
```

You will then be taken a web preview:



Kubernetes Dashboard

● Kubeconfig

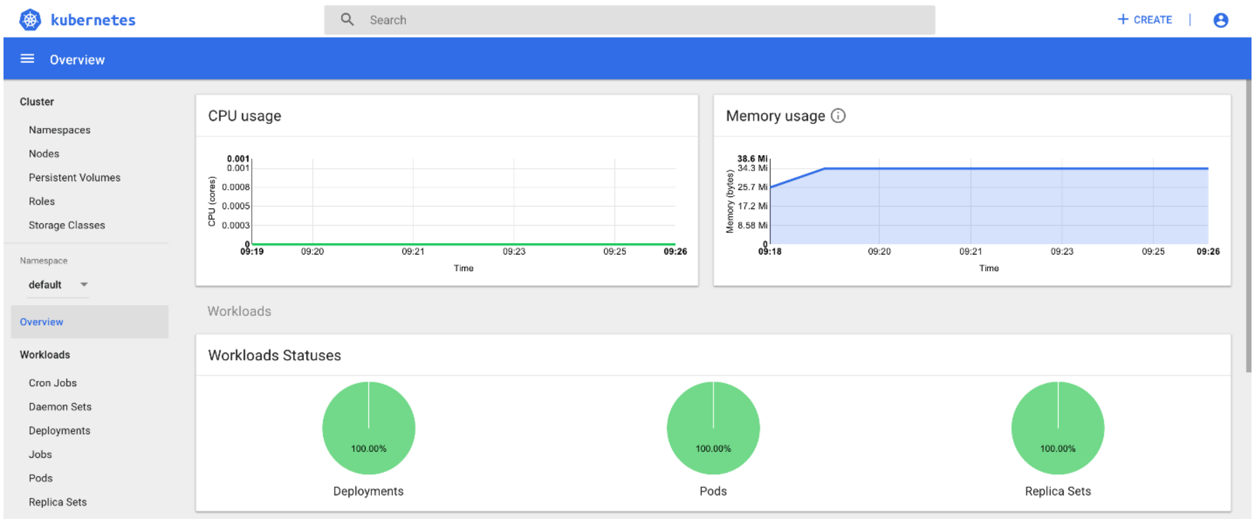
Please select the kubeconfig file that you have created to configure access to the cluster. To find out more about how to configure and use kubeconfig file, please refer to the Configure Access to Multiple Clusters section.

○ Token

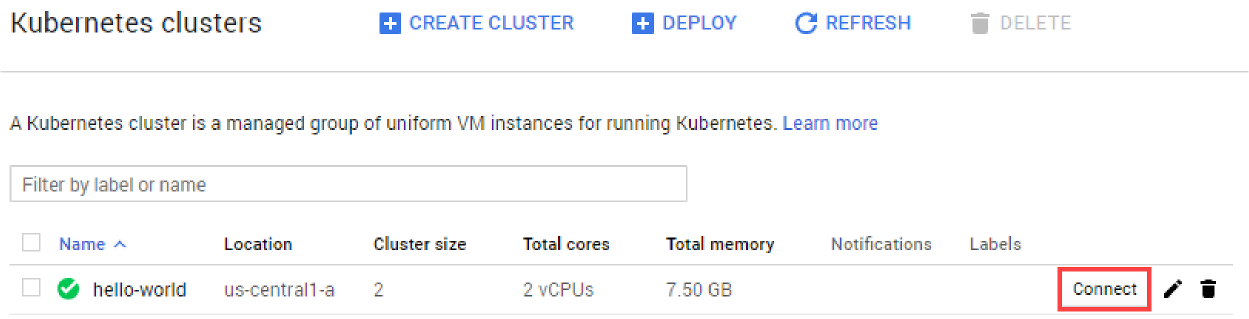
Every Service Account has a Secret with valid Bearer Token that can be used to log in to Dashboard. To find out more about how to configure and use Bearer Tokens, please refer to the Authentication section.

Choose kubeconfig file *

Select the **Token** radio button and paste the token copied from previous step. Click **Sign In**.
Enjoy the Kubernetes graphical dashboard and use it for deploying containerized applications, as well as for monitoring and managing your clusters!



You can access the dashboard from a development or local machine from the Web console. You would select **Navigation menu** > **Kubernetes Engine**, and then click the **Connect** button for the cluster you want to monitor.

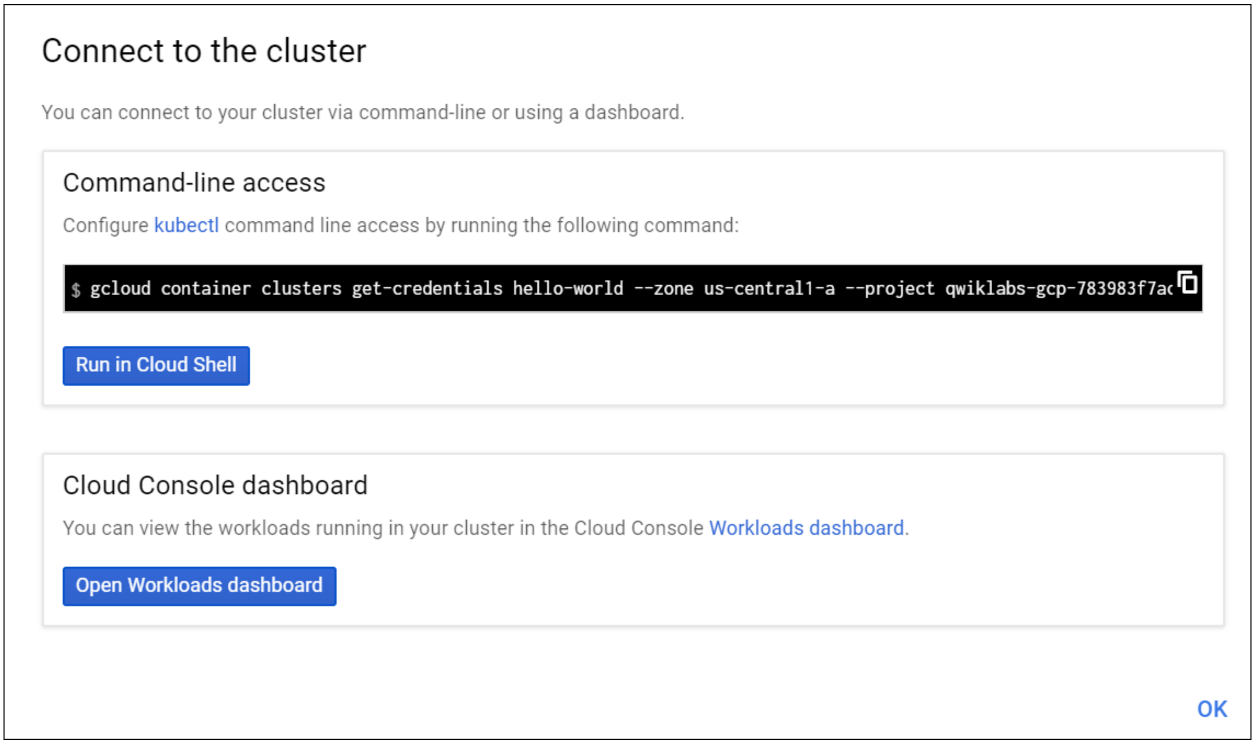


Kubernetes clusters

A Kubernetes cluster is a managed group of uniform VM instances for running Kubernetes. Learn more

Filter by label or name

□	Name ^	Location	Cluster size	Total cores	Total memory	Notifications	Labels	
□	✓ hello-world	us-central1-a	2	2 vCPUs	7.50 GB			Connect



Connect to the cluster

You can connect to your cluster via command-line or using a dashboard.

Command-line access

Configure kubectl command line access by running the following command:

```
$ gcloud container clusters get-credentials hello-world --zone us-central1-a --project qwiklabs-gcp-783983f7ac
```

Cloud Console dashboard

You can view the workloads running in your cluster in the Cloud Console Workloads dashboard.

Learn more about the Kubernetes dashboard by taking the Dashboard tour.