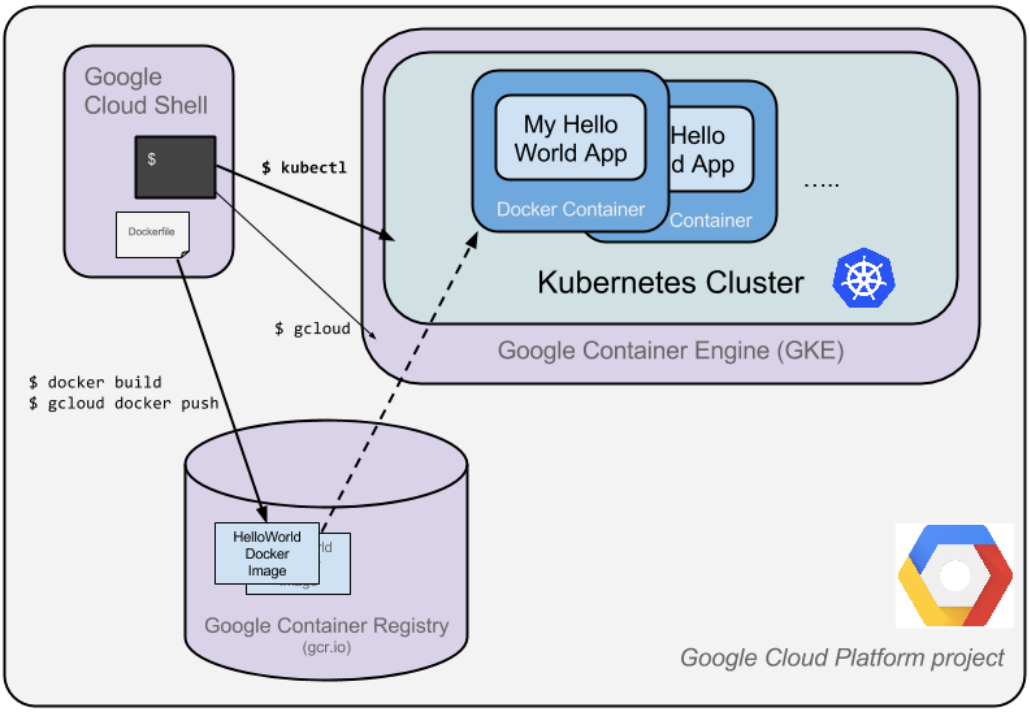
Hello Node Kubernetes

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.

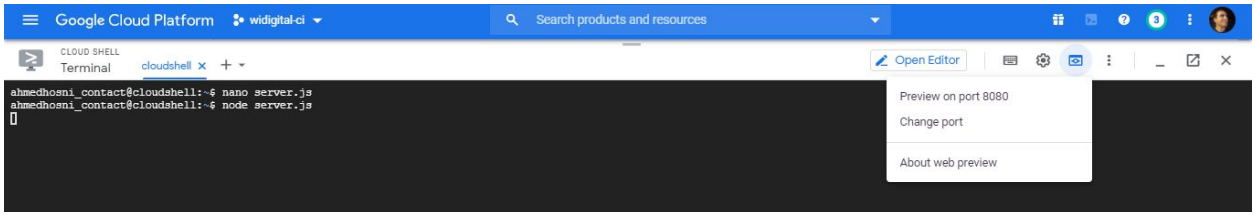
1- Create your Node.js application

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

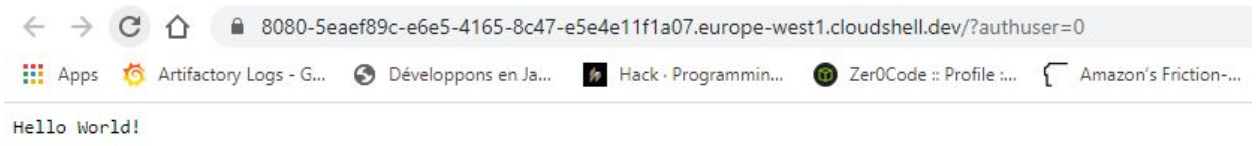
Since Cloud Shell has the `node` executable installed, run this command to start the node server (the command produces no output):

```
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:



2- 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.

```
nano Dockerfile
```

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.

Build the image with the following, replacing `PROJECT_ID` with your 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.

```
ahmedhosni_contact@cloudshell:~$ docker build -t gcr.io/widigital-ci/hello-node:v1 .
Sending build context to Docker daemon  109.8MB
Step 1/4 : FROM node:6.9.2
6.9.2: Pulling from library/node
75a822cd7888: Pull complete
57de64c72267: Pull complete
4306be1e8943: Pull complete
871436ab7225: Pull complete
0110c26a367a: Pull complete
1f04fe713f1b: Pull complete
ac7c0b5fb553: Pull complete
Digest: sha256:2e95be60faf429d6c97d928c762cb36f1940f4456ce4bd33fbdc34de94a5e043
Status: Downloaded newer image for node:6.9.2
 ---> faaadb4aaf9b
Step 2/4 : EXPOSE 8080
 ---> Running in 9f3301dc7b03
Removing intermediate container 9f3301dc7b03
 ---> 905230938789
Step 3/4 : COPY server.js .
 ---> 492b136dfd7d
Step 4/4 : CMD node server.js
 ---> Running in dcc758bb27a0
Removing intermediate container dcc758bb27a0
 ---> fa3dcc18ae07
Successfully built fa3dcc18ae07
Successfully tagged gcr.io/widigital-ci/hello-node:v1
```

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

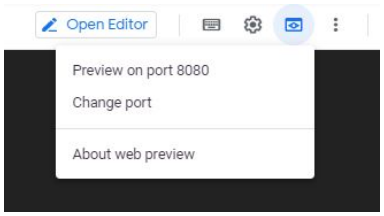
Run the following command replacing `PROJECT_ID` with your Project ID, found in the Console and the **Connection Details** section of the lab:

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

Your output should look something like this:

```
ahmedhosni_contact@cloudshell:~$ docker run -d -p 8080:8080 gcr.io/widigital-ci/hello-node:v1
33280191ca509db5a7bd5e347217d53788765c4c716dc64ae03efb62b4e1fee6
```

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



Or use `curl` from your Cloud Shell prompt:

```
ahmedhosni_contact@cloudshell:~$ curl http://localhost:8080
Hello World!ahmedhosni_contact@cloudshell:~$
```

Next, stop the running container.

Find your Docker container ID by running:

```
docker ps
```

Your output you should look like this:

```
ahmedhosni_contact@cloudshell:~$ docker ps
CONTAINER ID        IMAGE                                COMMAND                  CREATED             STATUS              PORTS                    NAMES
33280191ca50        gcr.io/widigital-ci/hello-node:v1    "/bin/sh -c 'node se…"   2 minutes ago       Up 2 minutes        0.0.0.0:8080->8080/tcp   heuristic_ellis
```

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):

```
ahmedhosni_contact@cloudshell:~$ docker stop 33280191ca50
33280191ca50
```

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 Project ID, found in the Console or the **Connection Details** section of the lab.

```
gcloud auth configure-docker
```

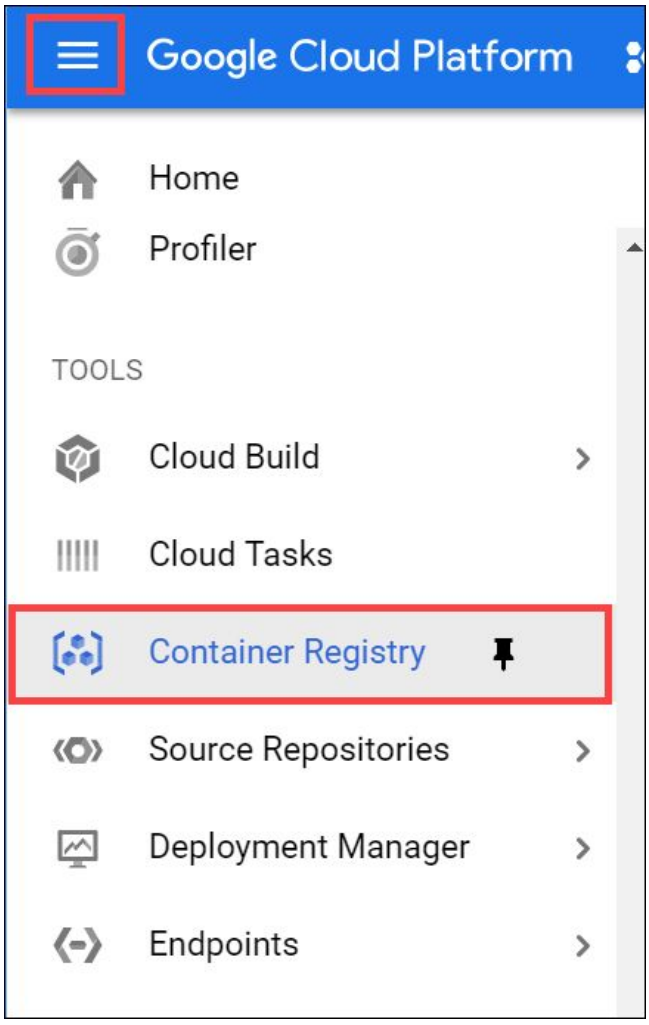
```
ahmedhosni_contact@cloudshell:~$ gcloud auth configure-docker
WARNING: Your config file at [/home/ahmedhosni_contact/.docker/config.json] contains these credential helper entries:

{
  "credHelpers": {
    "gcr.io": "gcloud",
    "us.gcr.io": "gcloud",
    "eu.gcr.io": "gcloud",
    "asia.gcr.io": "gcloud",
    "staging-k8s.gcr.io": "gcloud",
    "marketplace.gcr.io": "gcloud"
  }
}
Adding credentials for all GCR repositories.
WARNING: A long list of credential helpers may cause delays running 'docker build'. We recommend passing the registry name to configure only the registry you are using.
gcloud credential helpers already registered correctly.
```

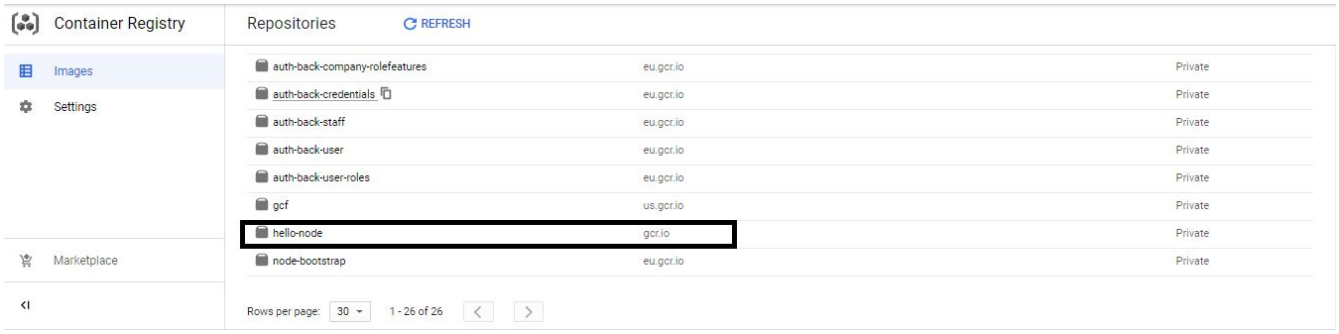
```
docker push gcr.io/PROJECT_ID/hello-node:v1
```

```
ahmedhosni_contact@cloudshell:~$ docker push gcr.io/widigital-ci/hello-node:v1
The push refers to repository [gcr.io/widigital-ci/hello-node]
19993a92e7bf: Pushed
381c97ba7dc3: Pushed
604c78617f34: Pushed
fa18e5ffd316: Pushed
0a5e2b2ddeaa: Pushed
53c779688d06: Pushed
60a0858edcd5: Pushed
b6ca02dfe5e6: Pushed
v1: digest: sha256:c77b8c848ba981fefac9215acbf7c3e5d903e03003e62bfe8562188cfde8a936 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.



3- 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 Project ID, found in the console and in the **Connection Details** section of the lab):

```
kubectl create deployment hello-node --image=gcr.io/PROJECT_ID/hello-node:v1
```

```
ahmedhosni_contact@cloudshell:~$ kubectl create deployment hello-node \
>     --image=gcr.io/widigital-ci/hello-node:v1
deployment.apps/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
```

```
ahmedhosni_contact@cloudshell:~$ kubectl get deployment
NAME                                          READY   UP-TO-DATE   AVAILABLE   AGE
hello-node                                    1/1     1            1           8m19s
incendiary-beetle-nginx-ingress-controller    1/1     1            1           42d
incendiary-beetle-nginx-ingress-default-backend   1/1     1            1           42d
spinnaker-1598656841-minio                    0/1     1            0           8h
```

To view the pod created by the deployment, run:

`kubectl get pods`

```
ahmedhosni_contact@cloudshell:~$ kubectl get pods
NAME                                READY   STATUS    RESTARTS   AGE
hello-node-78bffdb6f5-pst2x         1/1     Running   0          9m38s
```

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`

```
ahmedhosni_contact@cloudshell:~$ kubectl cluster-info
Kubernetes master is running at https://23.236.53.242
GLBCDefaultBackend is running at https://23.236.53.242/api/v1/namespaces/kube-system/services/default-http-backend:http/proxy
Heapster is running at https://23.236.53.242/api/v1/namespaces/kube-system/services/heapster/proxy
KubeDNS is running at https://23.236.53.242/api/v1/namespaces/kube-system/services/kube-dns:dns/proxy
Metrics-server is running at https://23.236.53.242/api/v1/namespaces/kube-system/services/https:metrics-server:/proxy

To further debug and diagnose cluster problems, use 'kubectl cluster-info dump'.
```

`kubectl config view`

```
ahmedhosni_contact@cloudshell:~$ kubectl config view
apiVersion: v1
clusters:
- cluster:
    certificate-authority-data: DATA+OMITTED
    server: https://35.241.213.150
  name: gke_widigital-ci_europe-west1-b_cluster-api
- cluster:
    certificate-authority-data: DATA+OMITTED
    server: https://23.236.53.242
  name: gke_widigital-ci_us-central1-f_cl-cluster
contexts:
- context:
    cluster: gke_widigital-ci_europe-west1-b_cluster-api
    user: gke_widigital-ci_europe-west1-b_cluster-api
  name: gke_widigital-ci_europe-west1-b_cluster-api
- context:
    cluster: gke_widigital-ci_us-central1-f_cl-cluster
    user: gke_widigital-ci_us-central1-f_cl-cluster
  name: gke_widigital-ci_us-central1-f_cl-cluster
current-context: gke_widigital-ci_us-central1-f_cl-cluster
kind: Config
preferences: {}
users:
- name: gke_widigital-ci_europe-west1-b_cluster-api
  user:
    auth-provider:
```

And for troubleshooting :

`kubectl get events`

```
ahmedhosni_contact@cloudshell:~$ kubectl config view
apiVersion: v1
clusters:
- cluster:
    certificate-authority-data: DATA+OMITTED
    server: https://35.241.213.150
  name: gke_widigital-ci_europe-west1-b_cluster-api
- cluster:
    certificate-authority-data: DATA+OMITTED
    server: https://23.236.53.242
  name: gke_widigital-ci_us-central1-f_cl-cluster
contexts:
- context:
    cluster: gke_widigital-ci_europe-west1-b_cluster-api
    user: gke_widigital-ci_europe-west1-b_cluster-api
ahmedhosni_contact@cloudshell:~$ kubectl get events
LAST SEEN   TYPE      REASON                  OBJECT                                  MESSAGE
7m17s       Normal    Killing                 pod/cd-redis-master-0                   Stopping container cd-redis
7m10s       Normal    Killing                 pod/cd-spinnaker-halyard-0              Stopping container halyard
6m3s        Normal    UPDATE                  ingress/esp-ingress                     Ingress default/esp-ingress
10m         Normal    CREATE                  ingress/esp-ingress                     Ingress default/esp-ingress
6m3s        Normal    UPDATE                  ingress/esp-ingress                     Ingress default/esp-ingress
11m         Warning   FailedScheduling        pod/hello-app-f99cf888-wjrmv            0/3 nodes are available: 3 Insufficient cpu.
10m         Normal    Pulling                 pod/hello-app-f99cf888-wjrmv            Pulling image "gcr.io/google-samples/hello-app:1.0"
10m         Normal    Pulled                  pod/hello-app-f99cf888-wjrmv            Successfully pulled image "gcr.io/google-samples/hello-app:1.0"
10m         Warning   Failed                  pod/hello-app-f99cf888-wjrmv            Error: cannot find volume "default-token-nsrn4" to mount into container "hello-app"
7m14s       Warning   FailedScheduling        pod/hello-node-78bffdb6f5-pst2x         0/3 nodes are available: 3 Insufficient cpu.
6m15s       Normal    Scheduled               pod/hello-node-78bffdb6f5-pst2x         Successfully assigned default/hello-node-78bffdb6f5-pst2x to gke-cl-cluster-default-pool-1240
```

`kubectl logs <pod-name>`

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

4- 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" --port=8080`

```
ahmedhosni_contact@cloudshell:~$ kubectl expose deployment hello-node --type="LoadBalancer" --port=8080
service/hello-node exposed
```

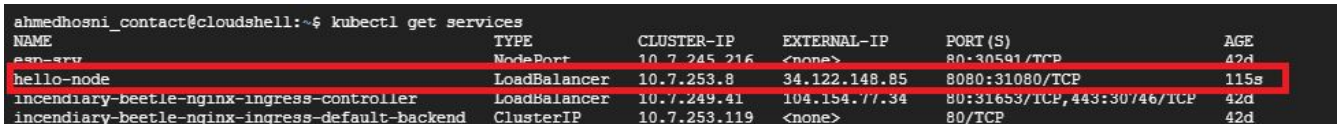
The flag used in this command specifies that are using the load-balancer provided by the underlying infrastructure (in this case the Compute Engine load balancer). Note that you expose the deployment, and not the pod, directly. This will cause the resulting

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.

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

```
kubectl get services
```

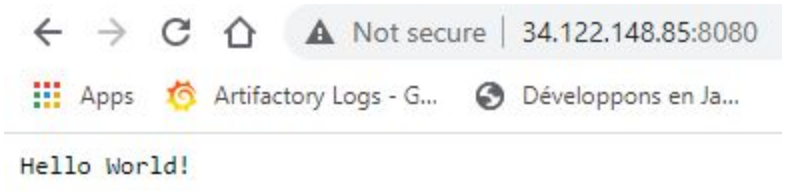


There are 2 IP addresses listed for your hello-node service, both serving port 8080. The `CLUSTER-IP` is the internal IP that is only visible inside your cloud virtual network; the `EXTERNAL-IP` is the external load-balanced IP.

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`



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.

5- 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
```

```
ahmedhosni_contact@cloudshell:~$ kubectl scale deployment hello-node --replicas=4
deployment.extensions/hello-node scaled
```

You can request a description of the updated deployment:

```
kubectl get deployment
```

```
ahmedhosni_contact@cloudshell:~$ kubectl get deployment
NAME                                          READY   UP-TO-DATE   AVAILABLE   AGE
hello-node                                    3/4     4            3           26m
incendiary-beetle-nginx-ingress-controller    1/1     1            1           42d
incendiary-beetle-nginx-ingress-default-backend   1/1     1            1           42d
spinnaker-1598656841-minio                    0/1     1            0           9h
```

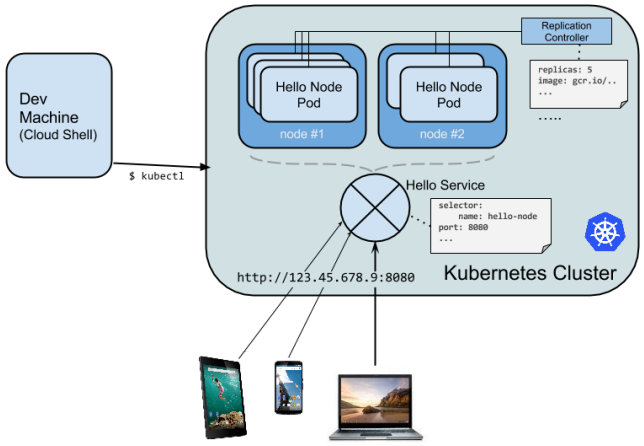
You may need to run the above command until you see all 4 replicas created. You can also list the all pods:

```
kubectl get pods
```

```
ahmedhosni_contact@cloudshell:~$ kubectl get pods
NAME                                                READY   STATUS    RESTARTS   AGE
hello-node-78bffdb6f5-6hcpj                         1/1     Running   0          5m56s
hello-node-78bffdb6f5-9pl7m                         1/1     Running   0          5m56s
hello-node-78bffdb6f5-pst2x                         1/1     Running   0          28m
hello-node-78bffdb6f5-r888d                         0/1     Pending   0          5m56s
incendiary-beetle-nginx-ingress-controller-569848bbd8-lgsvm   1/1     Running   0          42d
```

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:



6- 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`:

```
nano server.js
```

Then update the response message:

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

Save the `server.js` file

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 .
```

```
ahmedhosni_contact@cloudshell:~$ docker build -t gcr.io/widigital-ci/hello-node:v2 .
Sending build context to Docker daemon  109.8MB
Step 1/4 : FROM node:6.9.2
 ---> faaadb4aaf9b
Step 2/4 : EXPOSE 8080
 ---> Using cache
 ---> 905230938789
Step 3/4 : COPY server.js .
 ---> 00b464d46be4
Step 4/4 : CMD node server.js
 ---> Running in f7e8bc7c4a5c
Removing intermediate container f7e8bc7c4a5c
 ---> 247fbc072f28
Successfully built 247fbc072f28
Successfully tagged gcr.io/widigital-ci/hello-node:v2
```

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

```
ahmedhosni_contact@cloudshell:~$ docker push gcr.io/widigital-ci/hello-node:v2
The push refers to repository [gcr.io/widigital-ci/hello-node]
53ed3a4a33f4: Pushed
381c97ba7dc3: Layer already exists
604c78617f34: Layer already exists
fa18e5ffd316: Layer already exists
0a5e2b2ddeaa: Layer already exists
53c779688d06: Layer already exists
60a0858edcd5: Layer already exists
b6ca02dfe5e6: Layer already exists
v2: digest: sha256:860f4ff86adec6f77b970f69e974f25e04ce28ac6553d1986f61a2d0ed2180e4 size: 2002
```

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:

```
ahmedhosni_contact@cloudshell:~$ kubectl edit deployment hello-node
deployment.extensions/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 (you may need to rerun the above command to see the following):

```
ahmedhosni_contact@cloudshell:~$ kubectl get deployments
NAME                                          READY   UP-TO-DATE   AVAILABLE   AGE
hello-node                                    3/4     2            3           55m
incendiary-beetle-nginx-ingress-controller    1/1     1            1           42d
incendiary-beetle-nginx-ingress-default-backend   1/1     1            1           42d
spinnaker-1598656841-minio                    0/1     1            0           9h
```

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.

7- 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/recommended/kubernetes-dashboard.yaml
```

You should receive a similar output:

```
ahmedhosni_contact@cloudshell:~$ kubectl apply -f https://raw.githubusercontent.com/kubernetes/dashboard/v1.10.1/src/deploy/recommended/kubernetes-dashboard.yaml
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:

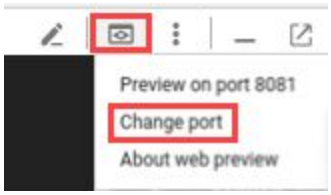
```
ahmedhosni_contact@cloudshell:~$ kubectl -n kube-system describe $(kubectl -n kube-system \
> get secret -n kube-system -o name | grep namespace) | grep token:
token:      eyJhbGciOiJSUzI1NiIsImtpZCI6IiJ9.eyJpc3MiOiJrdWJlcm5ldGVzL3NlcnZpY2VhY2NvdW50Iiwia3ViZXJuZXRlcy5pby9zZXJ2aWNlYWNjb3VudC9uYW1lc3BhY2UiOiJrdWJlLXN5c3RlbSIsImt1YmVybmV0ZXMuaW8vc2VydmljZWFjY291bnQvc2VjcmV0Lm5hbWUiOiJuYW1lc3BhY2UtY29udHJvbGxlci10b2tlbi9uYW1lc3BhY2UtY29udHJvbGxlciIsImt1YmVybmV0ZXMuaW8vc2VydmljZWFjY291bnQvc2VydmljZS1hY2NvdW50Lm5hbWUiOiJuYW1lc3BhY2UtY29udHJvbGxlciIsImt1YmVybmV0ZXMuaW8vc2VydmljZWFjY291bnQvc2VydmljZS1hY2NvdW50LnVpZCI6Ijg4MmIzNzY0LWM4NDYtMTFlYS04NzU1LTQyMDEwYTgwMDBiNSIsInN1YiI6InN5c3RlbTpzZXJ2aWNlYWNjb3VudDpkdWJlLXN5c3RlbTpuYW1lc3BhY2UtY29udHJvbGxlciJ9.sLhAyUv0Sj5yKUDFwLz1ytj19UY48oYc8CkVK-21lHnskqet004v3pyq8lty8k0XxmZSKGtn5SSx3r4Jt8P2lk0x6zP5pKzIF1nVXELKuoTHmd0wDYneQdolLYD04__fdOP_-WCKmdfT3ViFsxkhXmEqiqSWZmkQYyCn6p6VrffJDHg6b_fdS0Rx9CfTAOkko4Mvg1SDmK-uITbOku_JhqB0bXwhbFsA7uHibhZzlfLyhD7F1c4dFiJuYV8X85SdPNlshJ5YvCYtJdidq1P3n93VupBKYDRdJbTR_7uhSS7n-ZybR4Bac-vApYW9uXf3Ymj6NlYplaJon9N1-twofyg
```

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
```

```
ahmedhosni_contact@cloudshell:~$ kubectl proxy --port 8081
Starting to serve on 127.0.0.1:8081
```

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



This should send you to the API endpoint.

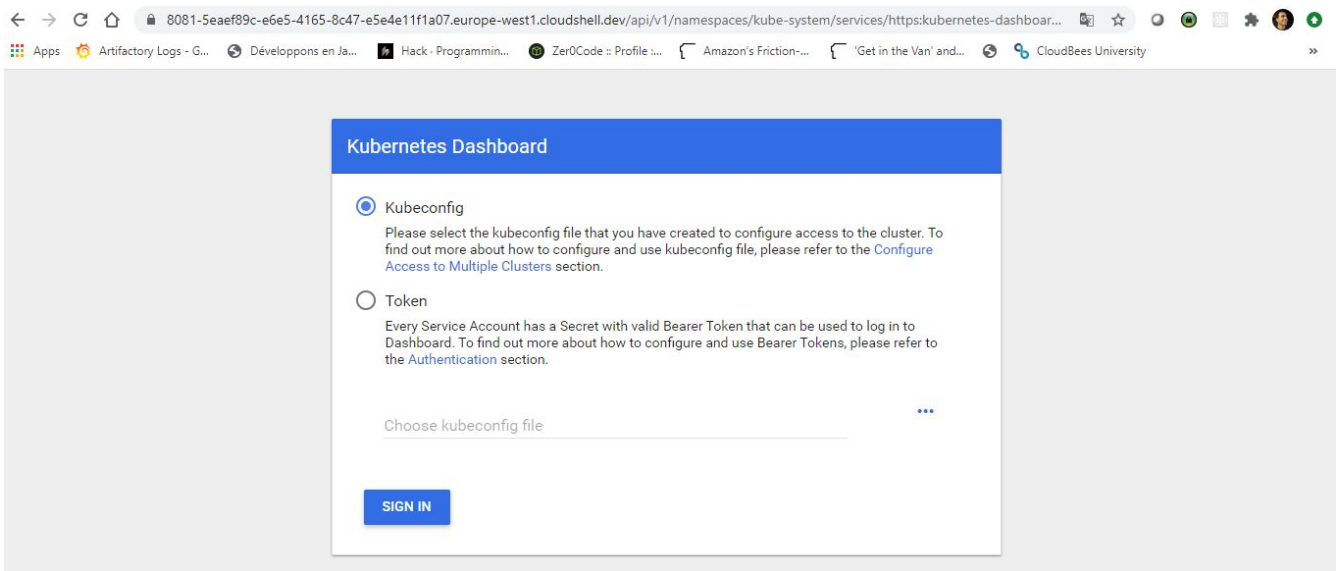
To get to the dashboard, remove `/?authuser=0` and append the URL with the following:

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

Your final URL should resemble the following:

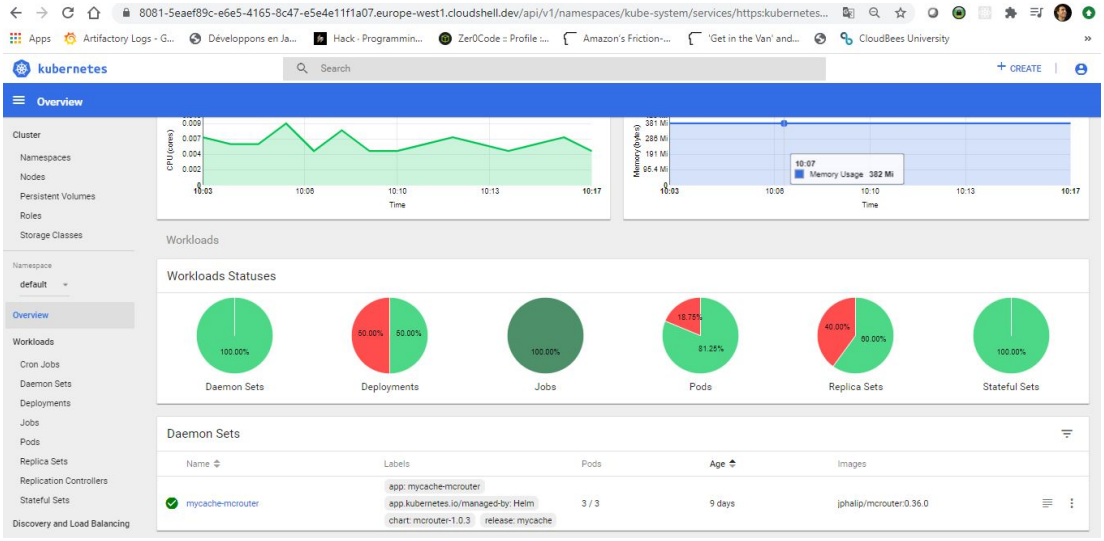
https://8081-5eaef89c-e6e5-4165-8c47-e5e4e11f1a07.europe-west1.cloudshell.dev/api/v1/namespaces/kube-system/services/https:kubernetes-dashboard:/proxy/#!/login

You will then be taken a web preview:

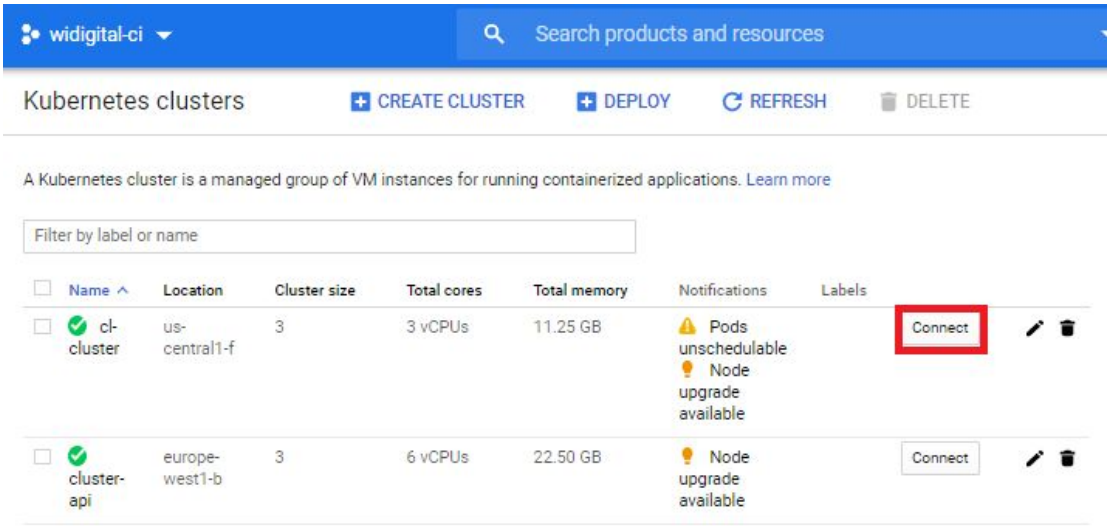


Select the Token radio button and paste the token copied from previous step (make sure the token is written in one single line). Click Sign In.

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.



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