

**Colin Dickerson**  
**CSYE 6225**  
**HW – Week 11**

## Setup

This assignment begins where the previous assignment (HW 10) left off. At this point we've created our first container object, and have our Node.JS project up and running & accessible. The next phase is to start using Kubernetes for cluster management. For this system we will again use the google cloud shell as our development/build machine and to interface into the cluster management software.

## Create your Kubernetes Cluster


To start we need to create a cluster environment for all of our containers to live in. Open the cloud console and type the following command to set the compute "zone". For my assignment I used the zone: "us-east1-b" since I am located on the east coast.

```
gcloud config set compute/zone us-east1-b
```


Next we can create the container cluster using a single command:

```
gcloud container clusters create hello-world
```


Once that's completed (It can take a few minutes to complete), we'll now have our own Container cluster to work in:



Container Engine




Container clusters




Container Registry


Container clusters



CREATE CLUSTER




REFRESH



DELETE

Container clusters

<input type="checkbox"/>	Name	Zone	Cluster size	Total cores	Total memory	Node version
<input type="checkbox"/>	 hello-world	us-east1-b	3	3 vCPUs	11.25 GB	1.4.6

Lastly, we need to configure the kubeconfig file with the proper credentials to connect to the cluster environment. This will also cause the *kubectl* command to interface to the

new cluster environment. Luckily google provides a simple command to complete this task:

```
gcloud container clusters get-credentials hello-world
```

## Create Your Pod

The next step is to create a Kubernetes pod. The pod is just a grouping of containers (or a single container), but it has the added benefit of allowing us to administrate multiple containers at once! Additionally, the pods add a layer of abstraction on top of the basic container object and are how Kubernetes works with containers.

To create the pod run the following command:

```
kubectl run hello-node --image=gcr.io/csye6225/hello-node:v1 --port=8080
```

This command also has the effect of creating a deployment to hold the pod. The deployment is a way of managing the scale of the application, so we can scale a deployment which will have the result of creating more pods to serve the application data (but more on that in a minute).

NOTE: My project ID is "csye6225" For your own project you need to change this value to YOUR project ID.

We can the run the command: `kubectl get deployments` to see the active deployments and `kubectl get pods` : to see the active pods

```
colin_dickerson@csye6225:~$ kubectl run hello-node --image=gcr.io/csye6225/hello-node:v1 --port=8080
deployment "hello-node" created
colin_dickerson@csye6225:~$ kubectl get deployments
NAME          DESIRED   CURRENT   UP-TO-DATE   AVAILABLE   AGE
hello-node    1         1         1            0           14s
colin_dickerson@csye6225:~$ kubectl get pods
NAME                                READY   STATUS    RESTARTS   AGE
hello-node-3621373899-dgoff        1/1     Running   0          2m
colin_dickerson@csye6225:~$
```

At this point we have a single deployment, and a single pod running in that deployment, but it's still not accessible to the outside world.

## Allow External Traffic

By default the pod is only accessible on the Kubernetes private “virtual” network. But we want to expose it to the public network. To do this we run the following command:

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

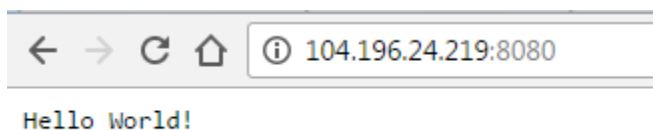
```
hello-node-3621373899-dgoff 1/1      Running    0          2m
colin_dickerson@csye6225:~$ kubectl expose deployment hello-node --type="LoadBalancer"
service "hello-node" exposed
colin_dickerson@csye6225:~$
```

The expose command is telling Kubernetes to make the previous deployment visible to the outside world. Note the type specification is “LoadBalancer” this is telling Kubernetes to have the exposed deployment balance load between the pods in the deployment. Now that the deployment is exposed we want to find out what the external (publically visible) IP address is. To find that out we need to run this command:

```
kubectl get services hello-node
```

```
colin_dickerson@csye6225:~$ kubectl get services hello-node
NAME         CLUSTER-IP   EXTERNAL-IP   PORT(S)    AGE
hello-node   10.3.240.249  104.196.24.219 8080/TCP    1m
colin_dickerson@csye6225:~$
```

Now we know the external IP of our service! If we access that IP address in a web browser over port 8080 we should see our “Hello World” service:

A screenshot of a web browser window. The address bar shows the URL '104.196.24.219:8080'. Below the address bar, the text 'Hello World!' is displayed in a simple, black, sans-serif font.

Which is exactly what we get! We now have our cluster environment configured and serving our application over a single IP address!

## Scale Up Your Website

At this point we have our cluster configured, but we only have a single pod running our site. Now we want to scale up our website to handle more traffic. To do this we only need to run a single command:

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

After running that command we can again run the “get deployment” and “get pods” commands to see that we now have a single deployment and 4 pods up and running. Note I had to run the “get pods” command twice since 2 of the 4 pods were still being spun up when I first ran the command.

```
colin_dickerson@csye6225:~$ kubectl scale deployment hello-node --replicas=4
deployment "hello-node" scaled
colin_dickerson@csye6225:~$ kubectl get deployment
NAME          DESIRED   CURRENT   UP-TO-DATE   AVAILABLE   AGE
hello-node    4         4         4            2          11m
colin_dickerson@csye6225:~$ kubectl get pods
NAME                                READY   STATUS             RESTARTS   AGE
hello-node-3621373899-14y2k        0/1     ContainerCreating   0          31s
hello-node-3621373899-dgoff        1/1     Running             0          11m
hello-node-3621373899-i0brh        1/1     Running             0          31s
hello-node-3621373899-vzxq8        0/1     ContainerCreating   0          31s
colin_dickerson@csye6225:~$ kubectl get pods
NAME                                READY   STATUS    RESTARTS   AGE
hello-node-3621373899-14y2k        1/1     Running   0          1m
hello-node-3621373899-dgoff        1/1     Running   0          12m
hello-node-3621373899-i0brh        1/1     Running   0          1m
hello-node-3621373899-vzxq8        1/1     Running   0          1m
colin_dickerson@csye6225:~$
```

This shows how Kubernetes can be configured to rapidly scale up and scale down. In more advanced configurations Kubernetes can be automatically configured to create/destroy pods based on the required load. The advantage of these pods/containers is that their footprint is so small that it takes only 1-2 minutes to spin up and configure a new pod, orders of magnitude faster than a traditional virtual machine

## Roll Out an Upgrade to Your Website

Finally, we want to simulate making an upgrade to our website. Any website will inevitable need bug fixes and addition of new features.

To begin we need to use our development machine (the cloud console) to edit the server.js file, and replace the line

```
response.end('Hello World!');
```

With

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

This can easily be achieved using nano or vim, whatever your preference!

Next we need to build a new version of the docker image:

```
docker build -t gcr.io/csye6225/hello-node:v2 .
```

and then push that new image to the cloud:

```
gcloud docker push gcr.io/csye6225/hello-node:v2
```

One thing to note is that this build/push cycle didn't take as long because we could utilize the docker cache to only build/push the necessary changes.

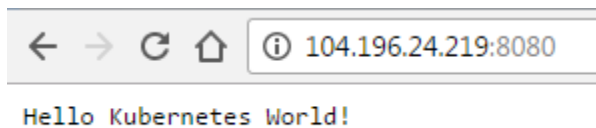
```
colin_dickerson@csye6225:~/hellonode$ docker build -t gcr.io/csye6225/hello-node:v2 .
Sending build context to Docker daemon 3.072 kB
Step 1 : FROM node:4.4
4.4: Pulling from library/node
d34921bc2709: Pull complete
7062b3d97728: Pull complete
767584930cea: Pull complete
c05d09cea848: Pull complete
fec753136a28: Pull complete
996bebbad3c5: Pull complete
3b60b36856ab: Pull complete
13a296c804e9: Pull complete
725242989af2: Pull complete
b2d5b3ebb221: Pull complete
Digest: sha256:7b657ccf24be2c8bab969b215e6853bc87a0d2fbc0896d5305cc87122f5360d0
Status: Downloaded newer image for node:4.4
--> b2d5b3ebb221
Step 2 : EXPOSE 8080
--> Running in 45db6b5046c2
--> 02a6317b5754
Removing intermediate container 45db6b5046c2
Step 3 : COPY server.js .
--> a63b26f817a3
Removing intermediate container b63a58d84eac
Step 4 : CMD node server.js
--> Running in 66772a97a477
--> 298febab3bbe
Removing intermediate container 66772a97a477
Successfully built 298febab3bbe
colin_dickerson@csye6225:~/hellonode$ gcloud docker push gcr.io/csye6225/hello-node:v2
WARNING: The '--' argument must be specified between gcloud specific args on the left and DOCKER_ARGS on the right.
. This usage is being deprecated and will be removed in March 2017.
This will be strictly enforced in March 2017. Use 'gcloud beta docker' to see new behavior.
Using 'push gcr.io/csye6225/hello-node:v2' for DOCKER_ARGS.
The push refers to a repository [gcr.io/csye6225/hello-node] (len: 1)
298febab3bbe: Pushed
a63b26f817a3: Pushed
02a6317b5754: Pushed
725242989af2: Image already exists
996bebbad3c5: Image already exists
fec753136a28: Image already exists
c05d09cea848: Image already exists
767584930cea: Image already exists
d34921bc2709: Image already exists
v2: digest: sha256:753e7d2c4271bcb527e05f68b5f440da3c38f734c4b819381ade904ff1572919 size: 12987
colin_dickerson@csye6225:~/hellonode$
```

Next, we need to point Kubernetes to the new docker image. To do this we need to run the following command:

```
kubectl set image deployment/hello-node hello-node=gcr.io/csye6225/hello-node:v2
```

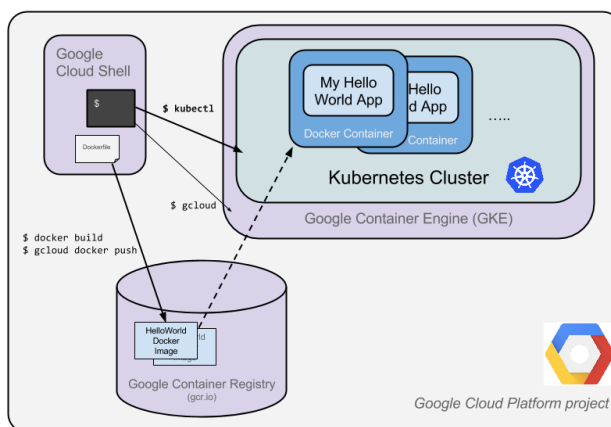
```
colin_dickerson@csye6225:~/hellonode$ kubectl set image deployment/hello-node hello-node=gcr.io/csye6225/hello-node:v2
deployment "hello-node" image updated
colin_dickerson@csye6225:~/hellonode$ kubectl get deployments
NAME          DESIRED   CURRENT   UP-TO-DATE   AVAILABLE   AGE
hello-node    4         4         4            4           27m
colin_dickerson@csye6225:~/hellonode$
```

Now if we access the same URL as in the “Allow External Traffic” section, we can see the webpage has been updated:



## Observations

Through this assignment we learned how to scale up the environment to handle additional load, and how we can easily deploy new changes to that environment. The process for updating the site is relatively straightforward:



First we make our changes and create a new Docker image, this is what will actual run our server.js code, making sure that we tag the image as a new version. Next we deployed the docker image to the cloud container repository. The container repository

houses the current and previous version of our container image. Now we have a newly published image, but the site is still running off of our older image. The last step is to tell kubernetes to upgrade our deployment to the new image version. After that kubernetes handles destroying the old pods and spinning up new pods. This really demonstrates how the container environment allows seamless upgrades to our site without us having to shut down the system or put our application into a “maintenance mode”

While the changes take a little bit longer to implement than just modifying a file on a standard web server, it’s not that much more complicated, and better yet, the Kubernetes system handles the updating of all of the scaled containers. For larger systems with hundreds or even thousands of containers, the ability to quickly scale and update these containers is immensely valuable. Current application design emphasizes continuous integration & deployment, thus having a system that can be easily (and possibly automatically) updating thousands of containers is a huge benefit of the cloud environment over a traditional web server system.

### **BONUS - Modify your server.js to include access persistence storage or volumes**

Next we’re going to modify and publish a new version of the server.js file that accesses persistent storage. For my example I chose to use MongoDB for the persistent storage engine.

To begin set the variable PROJECT\_ID to your actual project ID using the command:

```
export PROJECT_ID=<YOUR PROJECT ID HERE>
```

The first step in upgrading our site is to modify the server.js file to have a bit more code:

```

1  var http = require('http');
2  var handleRequest = function(request, response) {
3      console.log('Received request for URL: ' + request.url);
4      response.writeHead(200);
5      if (request.url !== '/') {
6          response.end();
7          return;
8      }
9
10     var MongoClient = require('mongodb').MongoClient;
11     var url = 'mongodb://mongo:27017/test';
12
13     MongoClient.connect(url, function(err, db) {
14         response.write('<html>');
15         response.write('<body>');
16         response.write('<h1>Hello World! HW11 - Colin Dickerson</h1>');
17
18         if (err !== null) {
19             response.write('<p>');
20             response.write(err.toString());
21             response.write('</p>');
22             response.end();
23         }
24         else {
25             var document = {action:"Hit", title:"New Hit Recorded"};
26             var collection = db.collection("hitcounter_collection_no_safe");
27
28             collection.insert(document, {w: 1}, function(ierr, records) {
29                 collection.count({}, function (error, count) {
30                     response.write('<p>Connected to Mongo Database! Current Hit Count:<b>');
31                     response.write(count.toString());
32                     response.write('</b></p></body></html>');
33                     response.end();
34                     db.close();
35                 });
36             });
37         }
38     });
39 };
40
41 var www = http.createServer(handleRequest);
42 www.listen(8080);
43

```

**NOTE:** You won't be able to test the server.js code in the cloud shell, like the previous file since the mongo db instance will only be accessible from other pods in the cluster. To test the server script I recommend spinning up a new, compute engine VM and installing a local instance of the MongoDB server and Node.JS. From there you can debug/test your script, and once it's up and running, copy it back over into the cloud shell environment for publishing. Just make sure you adjust the MongoDB URL field to the appropriate location. For local testing the URL is: *mongodb://localhost:27017/test*. For the Cluster environment the URL is: *mongodb://mongo:27017/test*.



Next we need to modify the Docker file to include the line: *npm install mongodb --save*. This will make sure the new Docker image gets the libraries it needs to connect to the mongo database server:

```
GNU nano 2.2.6
FROM node:4.4
EXPOSE 8080
COPY server.js .

RUN npm install mongodb --save

CMD node server.js
```

Next we need to generate a new “version 3” of the Docker image and push that to the container repository. Note that we won’t update the Kurbenetes deployment right now. Instead we’ll wait until the MongoDB system has been setup.

```
docker build -t gcr.io/$PROJECT_ID/hello-node:v3 .
gcloud docker push gcr.io/$PROJECT_ID /hello-node:v3
```

Now we need to create the MongoDB pods and push that out to Kurbenetes. However, for this installation to be persistent we need a place for Kurbenetes to store the data. To do this we need to create a persistent disk in the cloud environment using the command:

```
gcloud compute disks create --project $PROJECT_ID --zone "us-central1-a" -
-size 200gb mongo-disk
```

After creating the disk we need create a MongoDB controller pod and then expose that as a service in Kurbenetes. To do this we will use two “YAML” files; one for the controller and one for the service. YAML files are an all in one configuration file that describes to Kurbenetes how to create and deploy all different types of containers objects. This allows us to spin up different types of containers/pods based on the configuration file. In fact these configuration files can be combined to allow for a whole Kurbenetes environment to be configured and loaded using a single command (very handy for scripting the environment). In our case, to keep things simple, we’ll use two separate files.

```
GNU nano 2.2.6 File:
# db-controller.yml
apiVersion: v1
kind: ReplicationController
metadata:
  labels:
    name: mongo
  name: mongo-controller
spec:
  replicas: 1
  template:
    metadata:
      labels:
        name: mongo
    spec:
      containers:
      - image: mongo
        name: mongo
        ports:
        - name: mongo
          containerPort: 27017
          hostPort: 27017
        volumeMounts:
        - name: mongo-persistent-storage
          mountPath: /data/db
      volumes:
      - name: mongo-persistent-storage
        gcePersistentDisk:
          pdName: mongo-disk
          fsType: ext4

GNU nano 2.2.6
# db-service.yml
apiVersion: v1
kind: Service
metadata:
  labels:
    name: mongo
  name: mongo
spec:
  ports:
  - port: 27017
    targetPort: 27017
  selector:
    name: mongo
```

**NOTE:** in the db-controller.yml file the value for the "pdName" field is the same name as the disk we created previously ("mongo-disk"). Additionally, you can see that it's being built off a standard mongo image, so we don't need to worry about installing & configuring the database on our own. This is a great benefit of the cloud environment.

Next we want to make sure we're authenticated with the right cluster. This may not be necessary if you're running these commands immediately after creating the original Kubernetes cluster, but we'll run it again just to be safe. Run the command:

```
gcloud container clusters get-credentials hello-world
```

Note, the name of my cluster is "hello world", your name might be different. Now we'll create the MongoDB controller using the "create" command:

```
kubectl create -f db-controller.yaml
```

Then, we'll create the service using the same command but pointing to the service file

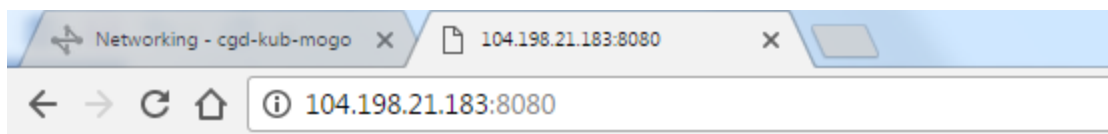
```
kubectl create -f db-service.yaml
```

Once the pods have finished spinning up, we can access the website using the "EXTERNAL-IP" address:

```
colin_dickerson@cgd-kub-mogo:~$ kubectl get pods
NAME                                READY   STATUS    RESTARTS   AGE
hello-node-2807417287-80p1o        1/1     Running   0           5h
hello-node-2807417287-1p5hg        1/1     Running   0           5h
hello-node-2807417287-qpf2y        1/1     Running   0           5h
hello-node-2807417287-skypy        1/1     Running   0          12h
mongo-controller-pje5t             1/1     Running   0          14h
colin_dickerson@cgd-kub-mogo:~$ kubectl get services hello-node
NAME            CLUSTER-IP   EXTERNAL-IP   PORT(S)    AGE
hello-node      10.43.252.182 104.198.21.183 8080/TCP   15h
colin_dickerson@cgd-kub-mogo:~$
```

When we access the page now we'll see a new page with an old fashioned Hit Counter. The code in server.js simply stores a timestamp record in a table every time the page is accessed. Then the website retrieves the row count and displays that on the screen.

We can test this to know it's working by deleting all of the pods, and waiting for the system to spin up new pods. When we do that we still get the persisted hit count, verifying the persistence engine is working:



## Hello World! HW11 - Colin Dickerson

Connected to Mongo Database! Current Hit Count:10

### BONUS Observations:

In this bonus section we added a new mongo pod to the cluster and exposed the service. From there the web service pods could access and update the data stored in the persistent data store.

One thing to keep in mind is that, while scaling the web server portion of the site is relatively straight forward: *"kubectl scale deployment hello-node --replicas=4"*, scaling of the database engine is not as simple, but it is possible according to the information I've read about the MongoDB system.