# Kubernetes (on Container Engine) - Basics to Advanced

Self-link: bit.ly/k8s-lab

Author: Ray Tsang @saturnism

## Introduction

Duration: 5:00

Hello everyone, thanks for coming today! Ready to learn Kubernetes? You will first become familiar with Compute Engine before working through a example Guestbook application, and then move on to more advanced Kubernetes experiments.

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 codelab, using a managed environment such as Google Container 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 but you should feel free to use your favorite environment instead.

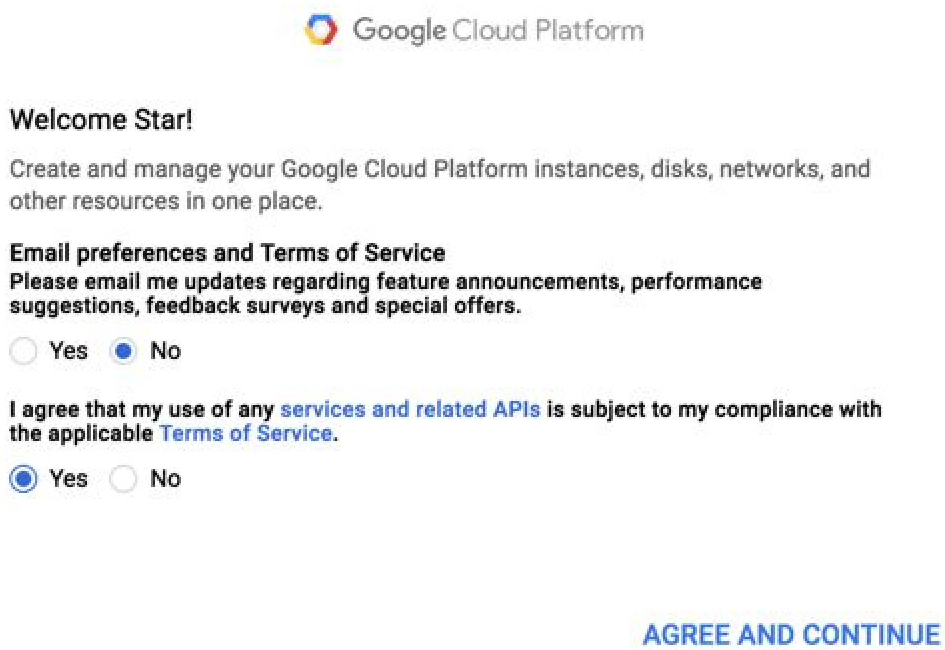
**What is your experience level with Containers?**

- I have just heard of Docker
- I played around with Docker
- I have containers in production
- I have already used container clustering technologies (kubernetes, mesos, swarm, ...)

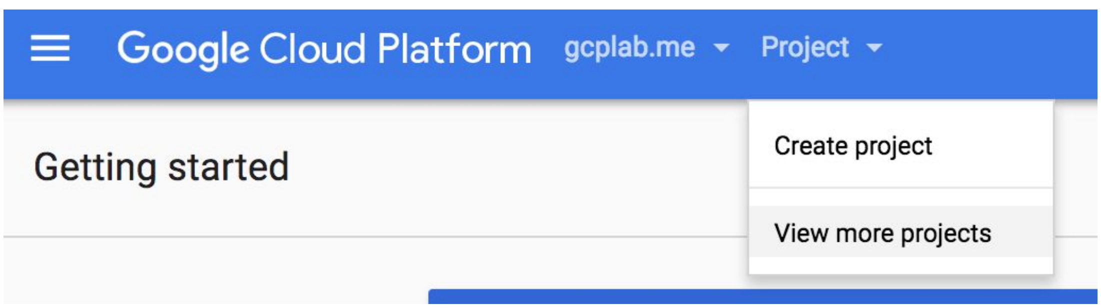
## Initial setup

Duration: 5:00

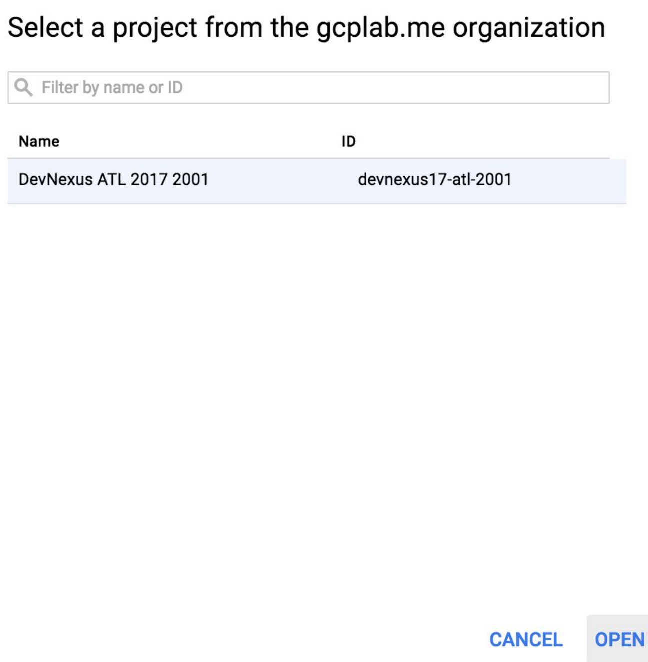
1. The instructor will provide you with a temporary username / password to login into Google Cloud Console.
2. To avoid conflicts with your personal account, please open a new incognito window for the rest of this lab.
3. Login into the Cloud Console: https://console.cloud.google.com/ with the provided credentials.
4. Accept the terms of service, and also configure the E-Mail preferences



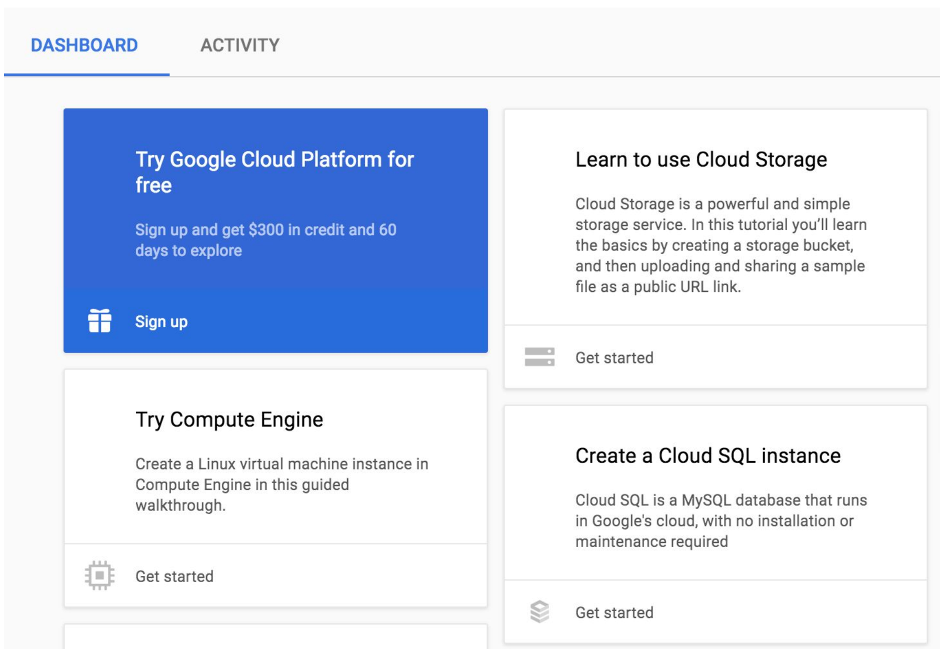

5. If you see a top bar with Sign Up for Free Trial written on it - DO NOT SIGN UP FOR THE FREE TRIAL. Click **Dismiss** since you'll be using a pre-provisioned lab account. If you are doing this on your own account, then you may want the free trial.
6. Make sure you click on the project `someproject-XXXX` that was pre-created for you - this is also your Project ID. Click **Project** > **View more projects**



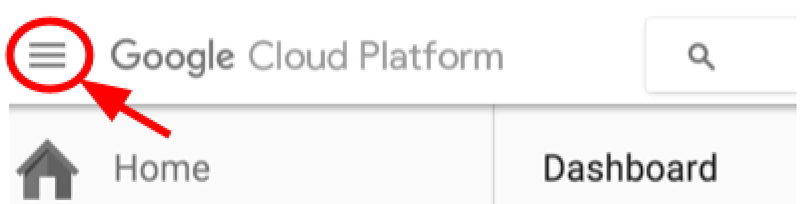

7. Select the pre-created project and click **Open**.

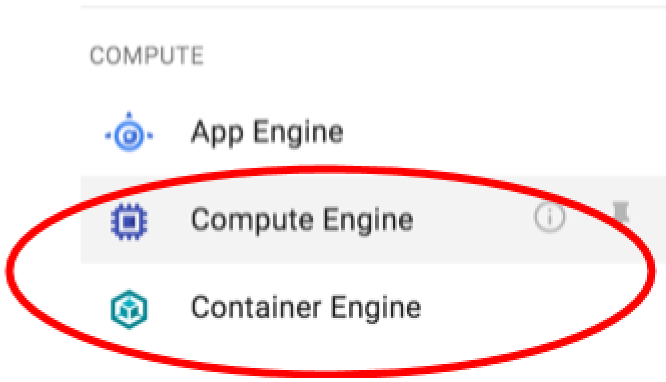



8. Once selected the project, you should see the Dashboard page




9. **Very Important -** Visit each of these pages to kick-off some initial setup behind the scenes, such as enabling the Compute Engine API by navigating to **Compute** → **Compute Engine** → **VM Instances**

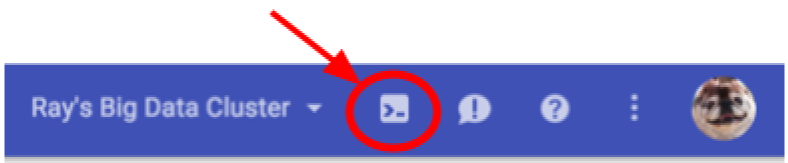




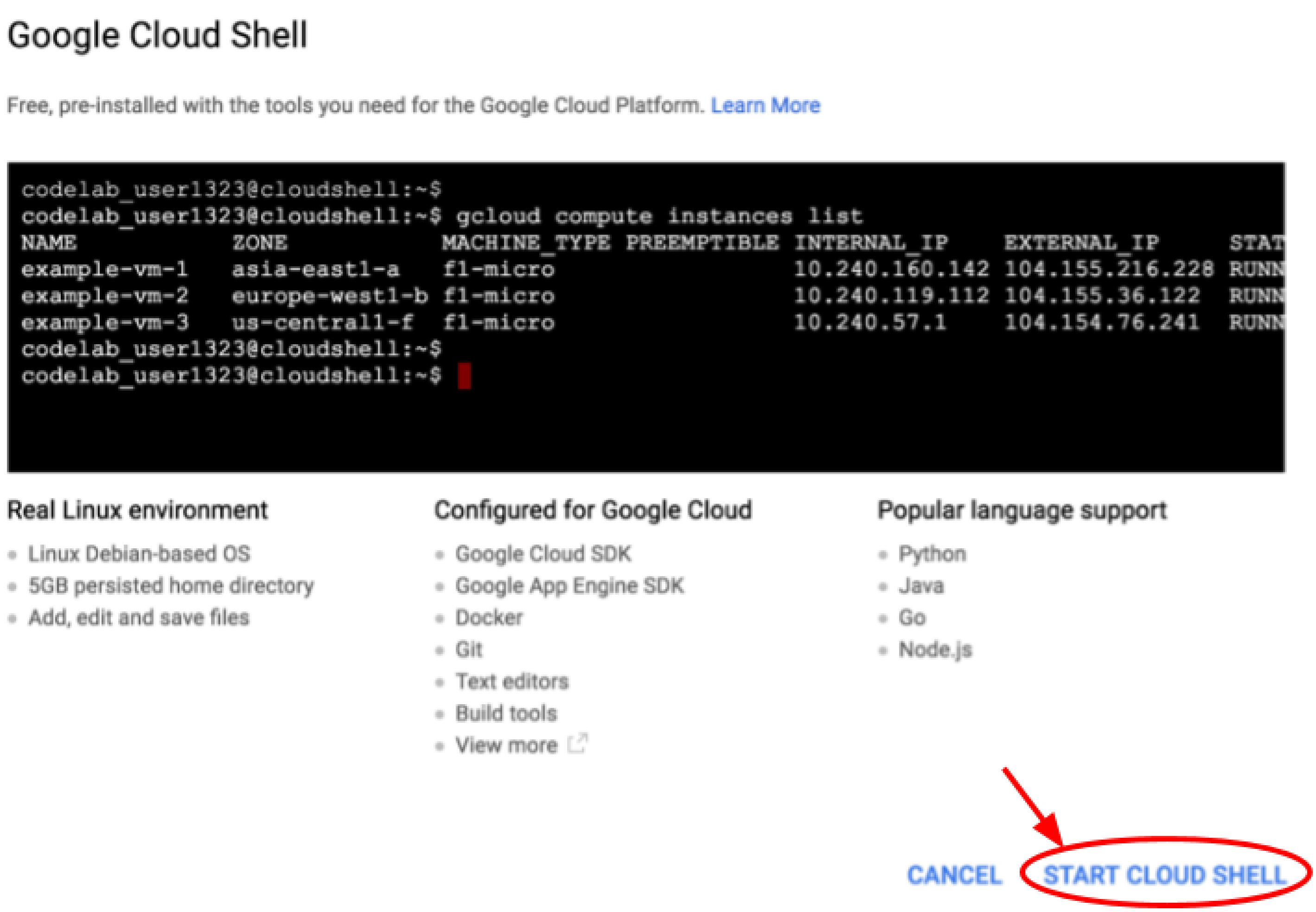

10. You will see the "Compute Engine is getting ready" message. Wait until this is done.


11. Similarly, enable to Container Engine API by navigating to **Compute → Container Engine → Container cluster**

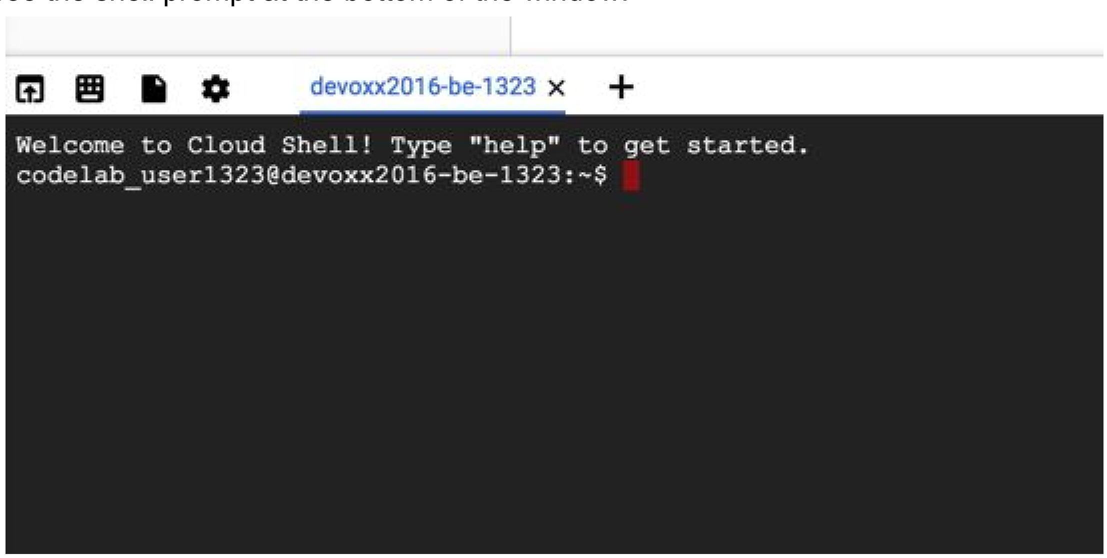
Once the operations completes, you will do most of the work from the Google Cloud Shell, a command line environment running in the Cloud. This Debian-based virtual machine is loaded with all the development tools you'll need (`docker`, `gcloud`, `kubectl` and others) and offers a persistent 5GB home directory. Open the Google Cloud Shell by clicking on the icon on the top right of the screen:




When prompted, click **Start Cloud Shell**:




You should see the shell prompt at the bottom of the window:




Finally, using Cloud Shell, set the default zone and project configuration:

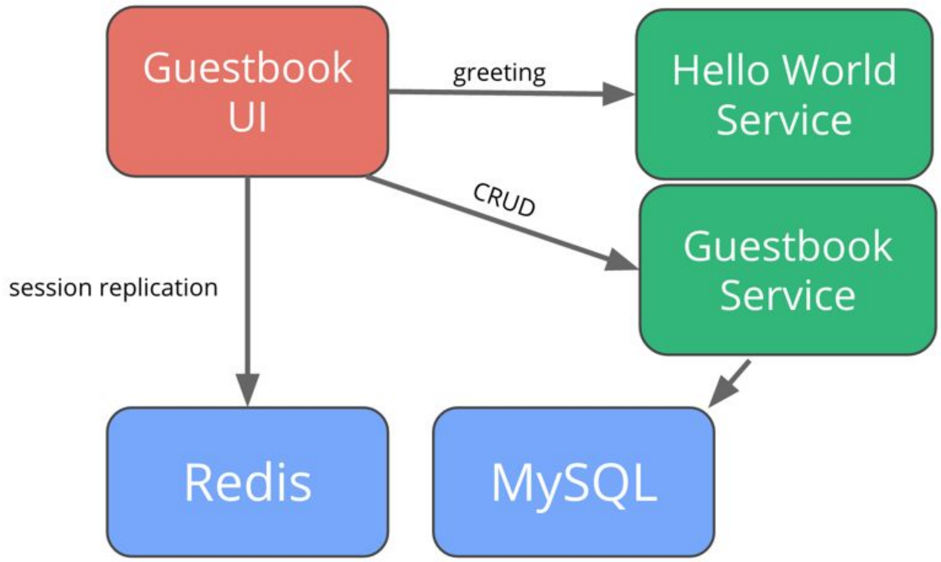
```
$ gcloud config set compute/zone europe-west1-c
$ gcloud config set compute/region europe-west1
```

You can pick and choose different zones too. Learn more about zones in [Regions & Zones documentation](Regions & Zones documentation).

**Note:** When you run `gcloud` on your own machine, the config settings would've been persisted across sessions. But in Cloud Shell, you will need to set this for every new session / reconnection.

## Kubernetes - The Basics

We're going to work through this [guestbook example](). This example is built with Spring Boot, with a frontend using Spring MVC and Thymeleaf, and two microservices. It requires MySQL to store guestbook entries, and Redis to store session information.




### Create your Kubernetes Cluster

Duration: 5:00

The first step is to create a cluster to work with. We will create a Kubernetes cluster using Google Container Engine.

Create a Kubernetes cluster in Google Cloud Platform is very easy! Use Container Engine to create a cluster:

```
$ gcloud container clusters create guestbook \
      --num-nodes 4 \
      --scopes cloud-platform
```

This will take a few minutes to run. Behind the scenes, it will create Google Compute Engine instances, and configure each instance as a Kubernetes node. These instances don't include the Kubernetes Master node. In Google Container Engine, the Kubernetes Master node is managed service so that you don't have to worry about it!

The scopes parameter is important for this lab. Scopes determine what Google Cloud Platform resources these newly created instances can access. By default, instances are able to read from Google Cloud Storage, write metrics to Google Cloud Monitoring, etc. For our lab, we add the `cloud-platform` scope to give us more privileges, such as writing to Cloud Storage as well.

While this goes on you might enjoy watching this short video https://youtu.be/7vZ9dRKRMyc!

You can see the newly created instances in the **Google Compute Engine** > **VM Instances** page.

## Run a

## Get the Guestbook source

Duration: 3:00

Start by cloning the GitHub repository for the Guestbook application:

```
$ git clone https://github.com/saturnism/spring-boot-docker
```

And move into the kubernetes examples directory.

```
$ cd spring-boot-docker/examples/kubernetes-1.2
```

We will be using the `yaml` files in this directory. Every file describes a resource that needs to be deployed into Kubernetes. Without giving much detail on its contents, but you are definitely encouraged to read them and see how pods, services, and others are declared. We'll talk a couple of these files in detail.

## Deploy Redis

Duration: 5:00

A Kubernetes pod is a group of containers, tied together for the purposes of administration and networking. It can contain one or more containers. All containers within a single pod will share the same networking interface, IP address, volumes, etc. All containers within the same pod instance will live and die together. It's especially useful when you have, for example, a container that runs the application, and another container that periodically polls logs/metrics from the application container.

You can start a single Pod in Kubernetes by creating a Pod resource. However, a Pod created this way would be known as a Naked Pod. If a Naked Pod dies/exits, it will not be restarted by Kubernetes. A better way to start a pod, is by using a higher-level construct such as Replication Controller, Replica Set, or a Deployment.

Prior to Kubernetes 1.2, Replication Controller is the preferred way deploy and manage your application instances. Kubernetes 1.2 introduced two new concepts - Replica Set, and Deployments.

Replica Set is the next-generation Replication Controller. The only difference between a Replica Set and a Replication Controller right now is the selector support. Replica Set supports the new set-based selector requirements whereas a Replication Controller only supports equality-based selector requirements.

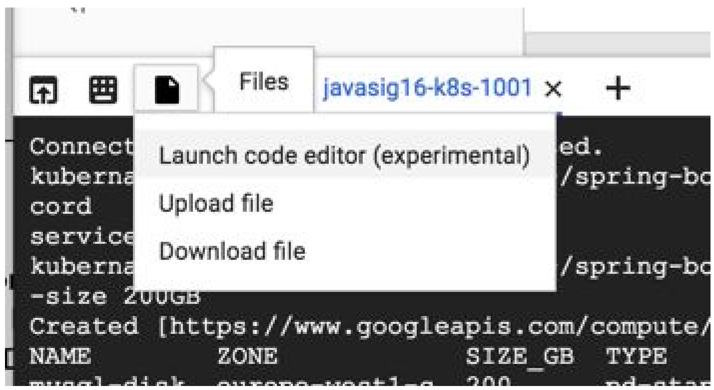
For example, Replication Controller can only select pods based on equality, such as "environment = prod", whereas Replica Sets can select using the "in" operator, such as "environment in (prod, qa)". Learn more about the different selectors in the Labels guide.

Deployment provides declarative updates for Pods and Replica Sets. You only need to describe the desired state in a Deployment object, and the Deployment controller will change the actual state to the desired state at a controlled rate for you. You can use deployments to easily:

- Create a Deployment to bring up a Replica Set and Pods.
- Check the status of a Deployment to see if it succeeds or not.
- Later, update that Deployment to recreate the Pods (for example, to use a new image, or configuration).
- Rollback to an earlier Deployment revision if the current Deployment isn’t stable.
- Pause and resume a Deployment.

In this lab, because we are working with Kubernetes 1.2+, we will be using Deployment extensively.

Open the `redis-deployment.yaml` to examine the deployment descriptor. You can use your favorite editor such as `vi`, `emacs`, or `nano`, but you can also use Cloud Shell's built-in code editor:




If you the Cloud Shell Code Editor, a new window will be opened, and you can navigate to open the file:


The instructors will explain the descriptor in detail. You can read more about Deployment in the [Kubernetes Deployment Guide](Kubernetes Deployment Guide).

First create a pod using `kubectl`, the Kubernetes CLI tool:

```
$ kubectl apply -f redis-deployment.yaml --record
```

You should see a Redis instance running:

```
$ kubectl get pods
NAME           READY     STATUS    RESTARTS    AGE
redis-....     1/1       Running   0           20s
```

**Optional interlude: Look at your pod running in a Docker container on the VM**
Kubernetes is container format agnostic. In your lab, we are working with Docker containers. Keep in mind that Kubernetes work with other container formats too. You can see that the Docker container is running on one of the machines. First, find the node name that Kubernetes scheduled this container to:

```
$ kubectl get pods -owide
NAME        READY     STATUS    RESTARTS    AGE        NODE
redis-...   1/1       Running   0           2m         gke-guestbook-...
```

The value under the label NODE is the name of the node.

You can then SSH into that node:

```
$ gcloud compute ssh <node-name>
```

```
WARNING: The private SSH key file for Google Compute Engine does not exist.
WARNING: You do not have an SSH key for Google Compute Engine.
WARNING: [/usr/bin/ssh-keygen] will be executed to generate a key.
This tool needs to create the directory [/home/kubernaut1119/.ssh]
before being able to generate SSH keys.

Do you want to continue (Y/n)? Y

Generating public/private rsa key pair.
Enter passphrase (empty for no passphrase): [Hit Enter]
Enter same passphrase again: [Hit Enter]
Your identification has been saved in /home/kubernaut1119/.ssh/google_compute_engine.
Your public key has been saved in /home/kubernaut1119/.ssh/google_compute_engine.pub.
The key fingerprint is:
…

someuser@<node-name>:~$
```

You can then use docker command line to see the running container:

```
someuser@<node-name>:~$ sudo docker ps
CONTAINER ID        IMAGE           COMMAND               CREATED              STATUS
67672e8118fd        redis:latest    "/entrypoint.sh       About an hour ago    Up
...
someuser@<node-name>:~$ exit
```

**End of Optional interlude: make sure you exit from the SSH** before you continue.

If you see other containers running don't worry, those are other services that are part of the management of Kubernetes clusters.

## Deploy a Redis Service

Duration: 3:00

Each Pod has a unique IP address - but the address is ephemeral.  The Pod IP addresses are not stable and it can change when Pods start and/or restart. A service provides a single access point to a set of pods matching some constraints. A Service IP address is stable.

Open the `redis-service.yaml` to examine the service descriptor. The important part about this file is the `selector` section. This is how a service knows which pod to route the traffic to, by matching the selector labels with the labels of the pods:

```
kind: Service
apiVersion: v1
metadata:
  name: redis
  labels:
    app: redis
```

```
    visualize: "true"
spec:
  ports:
    - port: 6379
      targetPort: 6379
  selector:
    app: redis
```

The instructors will explain the descriptor in detail. You can read more about Service in the Kubernetes Services Guide.

Create the Redis service:

```
$ kubectl apply -f redis-service.yaml --record
```

And check it:

```
$ kubectl get services
NAME         CLUSTER-IP      EXTERNAL-IP   PORT(S)    AGE
kubernetes   10.107.240.1    <none>        443/TCP    13m
redis        10.107.247.16   <none>        6379/TCP   52s
```

### Deploy MySQL and Service

Duration: 4:00

MySQL uses persistent storage. Rather than writing the data directly into the container image itself, our example stores the MySQL in a Google Compute Engine disk. Before you can deploy the pod, you need to create a disk that can be mounted inside of the MySQL container:

```
$ gcloud compute disks create mysql-disk --size 200GB
Created [...].
NAME       ZONE           SIZE_GB TYPE        STATUS
mysql-disk europe-west1-c 200     pd-standard READY
```

**Note:** If you see the message that the disk is not formatted - don't worry. It'll be formatted automatically when it's being used in the later step.

Open the `mysql-deployment.yaml` to examine the service descriptor. The important part about this file is the `volumes` and `volumeMounts` section. This is how a service knows which pod to route the traffic to, by matching the selector labels with the labels of the pods.

This section describes that the pod needs use a Google Compute Engine Persistent Disk that you created earlier, and also mounting that disk into a path specific to the MySQL container:

```
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  name: mysql
  labels:
    app: mysql
    visualize: "true"
spec:
  replicas: 1
  template:
    ...
    spec:
      containers:
      - name: mysql
        ...
        volumeMounts:
        - name: mysql-persistent-storage
          mountPath: /var/lib/mysql
      volumes:
      - name: mysql-persistent-storage
        gcePersistentDisk:
          # This GCE PD must already exist.
          pdName: mysql-disk
          fsType: ext4
```

The instructors will explain the descriptor in detail. You can read more about Volumes in the Kubernetes Volumes Guide.

You can then deploy both the MySQL Pod and the Service with a single command:

```
$ kubectl apply -f mysql-deployment.yaml -f mysql-service.yaml --record
```

Lastly, you can see the pods and service status via the command line. Recall the command you can use to see the status (hint: `kubectl get` ...). Make sure the status is `Running` before continuing.

In Kubernetes 1.4+, volume provision can be automatic in supported cloud platforms. Rather than creating a disk ahead of the time, you can simply say you need a volume - Kubernetes will automatically create the disk behind the scenes. See Kubernetes Persistent Volumes design.

You can find example of automatic provisioning in this repository: https://github.com/saturnism/spring-boot-docker/tree/master/examples/kubernetes-1.3.

## Deploy Microservices

Duration: 5:00

We have two separate services to deploy:

- the Guestbook service (that writes to the MySQL database)
- a Hello World service

Both services are containers whose images contain self-executing JAR files. The source is available in the `examples` directory if you are interested in seeing it.

When deploying these microservices instances, we want to make sure that:

- We can scale the number of instances once deployed.
- If any of the instances becomes unhealthy and/or fails, we want to make sure they are restarted automatically.
- If any of the machines that runs the service is down (scheduled or unscheduled), we need to reschedule the microservice instances to another machine.

Let's deploy the microservices one at a time:

First, deploy the Hello World:

```
$ kubectl apply -f helloworldservice-deployment-v1.yaml \
                -f helloworldservice-service.yaml \
                --record
```

Once created, you can see the replicas with:

```
$ kubectl get deployment
NAME                DESIRED   CURRENT   UP-TO-DATE   AVAILABLE   AGE
helloworld-service  2         2         2            0           28s
mysql               1         1         1            1           2m
redis               1         1         1            1           11m
```

You can see the pods running:

```
$ kubectl get pods
NAME                                  READY     STATUS              RESTARTS   AGE
helloworld-service-1726325642-fva3a   1/1       Running             0          1m
helloworld-service-1726325642-ujq2o   1/1       Running             0          1m
mysql-3871635011-82u7v                1/1       Running             0          2m
redis-2107737895-mnxr6                1/1       Running             0          11m
```

You can also look at each pod's log output by running:

```
$ kubectl logs -f helloworld-service-...
```

**Note:** The `-f` flag tails the log.  To stop tailing, press `Ctrl+C`.

The Deployment, behind the scenes, creates a Replica Set.  A Replica Set ensures the number of replicas (instances) you need to run at any given time. You can also see the Replica Set:

```
$ kubectl get rs
```

```
NAME                            DESIRED   CURRENT   AGE
helloworld-service-1726325642   2         2         2m
mysql-3871635011                1         1         4m
redis-2107737895                1         1         13m
```

Notice that because we also used Deployment to deploy both MySQL and Redis - each of those deployments created its own Replica Set as well.

Our descriptor file specified 2 replicas. So, if you delete one of the pods (and now you only have 1 replica rather than 2), the Replica Set will notice that and start another pod for you to meet the configured 2 replicas specification. Let's try it!

```
$ kubectl delete pod helloworld-service-...
pod "helloworldservice-..." deleted
```

You should see that the pod was deleted, and the Replication Controller will ensure a second instance is started. Sometimes this goes by very fast - and you'll notice that the pod you deleted is no longer there, and another pod, with a different name, was started.

```
$ kubectl get pods
NAME                                  READY     STATUS    RESTARTS   AGE
helloworld-service-1726325642-de1h4   1/1       Running   0          3s
helloworld-service-1726325642-ujq2o   1/1       Running   0          4m
...
```

Lastly, let’s create the Guestbook Service replication controller and service too!

```
$ kubectl apply -f guestbookservice-deployment.yaml \
                -f guestbookservice-service.yaml \
                --record
```

## A word on networking

Duration: 7:00

In Kubernetes every pod has a unique IP address!  You can “login” into one of these pods by using the `kubectl exec` command.  This can drop you into a shell and execute commands inside of the container.

First, find the name of the MySQL  pod:

```
$ kubectl get pod | grep mysql
mysql-...              1/1       Running   0          13m
```

Then, use `kubectl exec` to "login" into the container:

```
$ kubectl exec -ti mysql-... /bin/bash
```

```
root@mysql-...:/#
```

You are now in a shell inside of the MySQL container. You can run `ps`, and `hostname`:

```
root@mysql-...:/# ps auwx
USER       PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
mysql        1  0.0 12.3 994636 470492 ?       Ssl  20:32   0:01 mysqld
root       128  0.0  0.0  20224  3208 ?        Ss   21:09   0:00 /bin/bash
root       136  0.0  0.0  17488  2108 ?        R+   21:11   0:00 ps auwx

root@mysql-...:/# hostname -i
10.104.0.8

root@mysql-...:/# exit
```

Don't forget to exit :). Try it with another pod, like one of the Hello World Service pods and see its IP address.

```
$ kubectl exec -ti helloworld-service-... /bin/bash
root@helloworld-...:/app/src# hostname
helloworld-service-1726325642-de1h4
root@helloworld-...:/app/src# hostname -i
10.104.1.5
root@helloworld-...:/app/src# exit
```

Since we are running two instances of the Hello World Service (one instance in one pod), and that the IP addresses are not only unique, but also ephemeral - how will a client reach our services? We need a way to discover the service.

In Kubernetes, Service Discovery is a first class citizen. We created a Service that will:

- act as a load balancer to load balance the requests to the pods, and
- provide a stable IP address, allow discovery from the API, and also create a DNS name!

If you login into a container (find and use the Redis container), you can access the `helloworldservice` via the DNS name:

```
$ kubectl exec -ti redis-... /bin/bash
root@redis:/data# wget -qO- http://helloworld-service:8080/hello/Ray
{"greeting":"Hello Ray from helloworld-service-... with 
1.0","hostname":"helloworld-service-...","version":"1.0"}root@red
is:/data#
root@redis:/data# exit
```

Pretty simple right!?

## Deploy the Frontend

Duration: 5:00

You know the drill by now. We first need to create the replication controller that will start and manage the frontend pods, followed by exposing the service. The only difference is that this time, *the service needs to be externally accessible*. In Kubernetes, you can instruct the underlying infrastructure to create an external load balancer, by specifying the Service Type as a `LoadBalancer`.

You can see it in the `helloworldui-service.yaml`:

```
kind: Service
apiVersion: v1
metadata:
  name: helloworldui
  labels:
    name: helloworldui
    visualize: "true"
spec:
  type: LoadBalancer
  ports:
    - port: 80
      targetPort: http
  selector:
    name: helloworldui
```

Let’s deploy both the replication controller and the service at the same time:

```
$ kubectl apply -f helloworldui-deployment-v1.yaml \
                 -f helloworldui-service.yaml \
                 --record
```

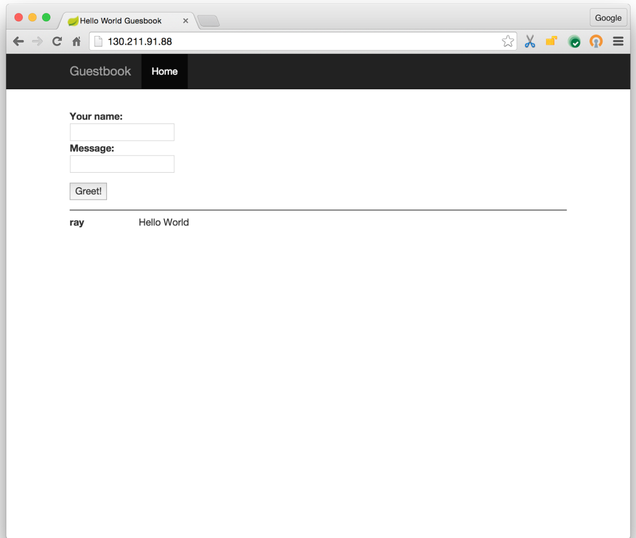
You can also access the public IP running, and look for `LoadBalancer Ingress IP` in the output in a minute or two:

```
$ kubectl describe services helloworld-ui
Name:                   helloworld-ui
Namespace:              default
Labels:                 name=helloworldui,visualize=true
Selector:               name=helloworldui
Type:                   LoadBalancer
IP:                     10.107.255.103
LoadBalancer Ingress:   X.X.X.X
Port:                   <unnamed>       80/TCP
NodePort:               <unnamed>       32155/TCP
Endpoints:              10.104.1.6:8080,10.104.1.7:8080
Session Affinity:       None
No events.
```

**Note:** The external load balancer may take a minute or two to create. Please retry the command above until the `LoadBalancer Ingress` shows up.

You can now access the guestbook via the ingress IP address by navigating the browser to `http://INGRESS_IP/`.

You should see something like this:




## Scaling In and Out

Duration: 5:00

Scaling the number of replicas of our Hello World service is as simple as running :

```
$ kubectl scale deployment helloworld-service --replicas=4
```

You can very quickly see that the replication controller has been updated:

```
$ kubectl get deployment
NAME                DESIRED    CURRENT    UP-TO-DATE    AVAILABLE    AGE
guestbook-service   2          2          2             2            18m
helloworld-service  4          4          4             2            24m
helloworld-ui       2          2          2             2            4m
mysql               1          1          1             1            26m
redis               1          1          1             1            35m
$ kubectl get pods
NAME                      READY     STATUS               RESTARTS    AGE
guestbook-service-...     1/1       Running              0           18m
guestbook-service-...     1/1       Running              0           18m
helloworld-service-...    0/1       ContainerCreating    0           25s
helloworld-service-...    1/1       Running              0           19m
helloworld-service-...    1/1       Running              0           24m
helloworld-service-...    0/1       ContainerCreating    0           25s
...
```

Let's scale out even more!

```
$ kubectl scale deployment helloworld-service --replicas=12
```

Let's take a look at the status of the pods:

```
$ kubectl get pods
NAME                                      READY     STATUS    RESTARTS   AGE
guestbook-service-3803699114-d73go    1/1       Running   0          21m
guestbook-service-3803699114-qnhsf    1/1       Running   0          21m
helloworld-service-1726325642-0yot3   1/1       Running   0          1m
helloworld-service-1726325642-2xlqg   1/1       Running   0          2m
helloworld-service-1726325642-4izw5   1/1       Running   0          1m
helloworld-service-1726325642-cal7t   1/1       Running   0          1m
helloworld-service-1726325642-de1h4   1/1       Running   0          22m
helloworld-service-1726325642-il7aj   0/1       Pending   0          1m
helloworld-service-1726325642-m901w   1/1       Running   0          1m
helloworld-service-1726325642-nz6py   1/1       Running   0          1m
helloworld-service-1726325642-ptdbp   0/1       Pending   0          1m
helloworld-service-1726325642-tftbh   1/1       Running   0          1m
helloworld-service-1726325642-ujq2o   1/1       Running   0          27m
helloworld-service-1726325642-z25ba   1/1       Running   0          2m
helloworld-ui-1131581392-qa1sy        1/1       Running   0          6m
helloworld-ui-1131581392-yyuuw        1/1       Running   0          6m
mysql-3871635011-82u7v                1/1       Running   0          28m
redis-2107737895-mnxr6                1/1       Running   0          37m
```

Oh no! Some of the pods are in the `Pending` state! That is because we only have four physical nodes, and the underlying infrastructure has run out of capacity to run the containers with the requested resources.

Pick a Pod name that is associated with the Pending state to confirm the lack of resources in the detailed status:

```
$ kubectl describe pod helloworld-service...
Name:                           helloworldui-service...
Namespace:                      default
Image(s):                       saturnism/spring-boot-helloworld-ui:v1
Node:                           /
Labels:                         name=helloworldui, ...
Status:                         Pending
...
Events:
  FirstSeen     LastSeen        Count   From                    SubobjectPath   Type
Reason                  Message
  ---------     --------        -----   ----                    -------------
--------        ------                  -------
  1m            1m              2       {default-scheduler }                    Warning
FailedScheduling        pod (helloworld-service-172632564
2-ptdbp) failed to fit in any node
fit failure on node (gke-guestbook-default-pool-8de71693-e0hx): Node didn't have enough
resource: CPU, requested: 100, used: 920, capacity: 1000
```

```
fit failure on node (gke-guestbook-default-pool-8de71693-6r1e): Node didn't have enough resource: CPU, requested: 100, used: 1000, capacity: 1000
```

The good news is that we can easily spin up another Compute Engine instance to append to the cluster. First, find the Compute Engine Instance Group that's managing the Kubernetes nodes (the name is prefixed with "`gke-`"). But you can resize the cluster simply from the command line:

```
$ gcloud container clusters resize guestbook --size=5
Pool [default-pool] for [guestbook] will be resized to 5.
Do you want to continue (Y/n)?  y
```

You can see a new Compute Engine instance is starting:

```
$ gcloud compute instances list
gke-guestbook-default-pool-3a020500-2t4q  europe-west1-c  n1-standard-1
10.240.0.3   130.211.64.214   RUNNING
...
gke-guestbook-default-pool-3a020500-t1ud  europe-west1-c  n1-standard-1
10.240.0.6   146.148.13.23    STAGING
...
```

Once the new instance has joined the Kubernetes cluster, you'll should be able to see it with this command:

```
$ kubectl get nodes
NAME                                 LABELS                           STATUS
gke-guestbook-a3e896df-node-3d99   kubernetes.io/hostname=...   Ready
gke-guestbook-a3e896df-node-dt8a   kubernetes.io/hostname=...   Ready
gke-guestbook-a3e896df-node-rqfg   kubernetes.io/hostname=...   Ready
gke-guestbook-a3e896df-node-vt3l   kubernetes.io/hostname=...   Ready
gke-guestbook-a3e896df-node-vt34   kubernetes.io/hostname=...   Ready
```

Use `kubectl get pods` to see the that the `Pending` pods have been scheduled.

Once you see they are scheduled, reduce the number of replicas back to 4 so that we can free up resources for the later labs:

```
$ kubectl scale deployment helloworld-service --replicas=4
```

## Rolling Update

Duration: 7:00

It's easy to update & rollback.

In this lab, we'll switch to the `examples/helloworld-ui` directory and make a minor change to the `templates/index.html` (e.g., change the background color, title, etc.). You can use your favorite editors, or use the Cloud Shell Code Editor that you used in the previous section.

After modifying the file, you'll rebuild the container and upload it to the [Google Container Registry](). You need to look up your project id by running `gcloud config list | grep pr oject`.

```
# Find the Project ID
$ gcloud config list | grep project

$ cd ~/spring-boot-docker/examples/helloworld-ui
$ docker build -t gcr.io/<your-project-id>/helloworld-ui:v2 .
$ gcloud docker -- push gcr.io/<your-project-id>/helloworld-ui:v2
```

**Note:** Because the Cloud Shell is running inside of a small VM instance it's not the fastest when it comes to extracting and buffering the container images! Once you start the push, it's a good time to take a break or … why not watching another video? This one about Google Container Registry: [https://www.youtube.com/watch?v=9CDb9ZSsfV4](https://www.youtube.com/watch?v=9CDb9ZSsfV4) !

Because we are managing our Pods with Deployment, it simplifies re-deployment with a new image and configuration. To use Deployment to update to Helloworld UI 2.0, first, edit the Deployment:

```
$ kubectl edit deployment helloworld-ui --record
```

**Note:** By default this is going to use `vi` as the editor! If you don't want to use `vi`, you can set the environmental variable `EDITOR` or `KUBE_EDITOR` to point to your favorite editor. E.g., `EDITOR=nano kubectl edit deployment helloworld-ui .`

You can then edit the Deployment directly in the editor. Change the image to the one you just created:

```
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  annotations:
    deployment.kubernetes.io/revision: "3"
    kubernetes.io/change-cause: kubectl edit deployment helloworld-ui --record
…
spec:
…
  template:
    …
    spec:
      …
      containers:
      - image: gcr.io/<your-project-id>/helloworld-ui:v2
      ...
```

Save and exit the editor, and you should see the message:

```
deployment "helloworld-ui" edited
```

That's it! Kubernetes will then perform a rolling update to update all the versions from 1.0 to 2.0.

In this lab, we use `kubectl edit` extensively to interactively update descriptors. You can use `kubectl patch` or `kubectl set` commands to make updates too. This is great for scripting the updates.

For example, to update the image, you can do:

```
kubectl set image deployment/helloworld-ui \
  helloworld-ui=gcr.io/<your-project-id>/helloworld-ui:v2
```

## Rollback a Deployment

Duration: 10:00

You can see your deployment history:

```
$ kubectl rollout history deployment helloworld-ui
REVISION        CHANGE-CAUSE
1               <none>
2               kubectl edit deployment helloworld-ui --record
```

Because when we edited the Deployment, we supplied the `--record` argument, the Change Cause value is automatically recorded with the command line that was executed.

You can rollback a Deployment to a previous revision:

```
$ kubectl rollout undo deployment helloworld-ui
deployment "helloworld-ui" rolled back
```

## Kubernetes Dashboard

You can access the Kubernetes Dashboard that is deployed inside of the cluster (by default). First, you need to get the login credential to the cluster. Use `gcloud container clusters describe` command to get detailed information about your cluster in Google Container Engine, and look for the username and password:

```
$ gcloud container clusters describe guestbook
...
masterAuth:
  password: ...
  username: admin
...
```

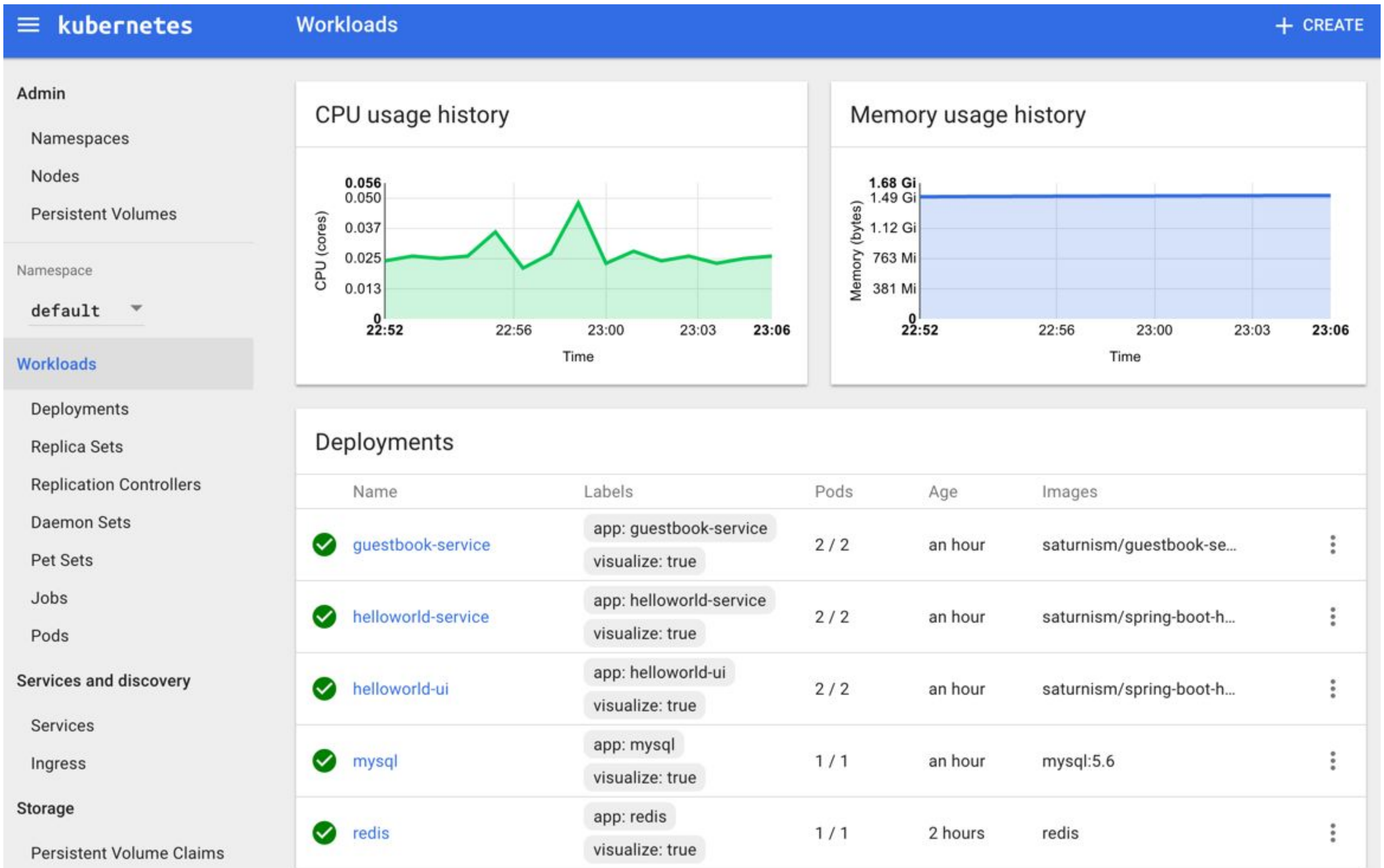
Note down the username and password.

Next, find the URL to Kubernetes Dashboard using the `kubectl cluster-info` command:

```
$ kubectl cluster-info
Kubernetes master is running at https://...
```

```
GLBCDefaultBackend is running at
https://.../api/v1/proxy/namespaces/kube-system/services/default-http-backend
Heapster is running at
https://.../api/v1/proxy/namespaces/kube-system/services/heapster
KubeDNS is running at https://.../api/v1/proxy/namespaces/kube-system/services/kube-dns
kubernetes-dashboard is running at
https://.../api/v1/proxy/namespaces/kube-system/services/kubernetes-dashboard
```

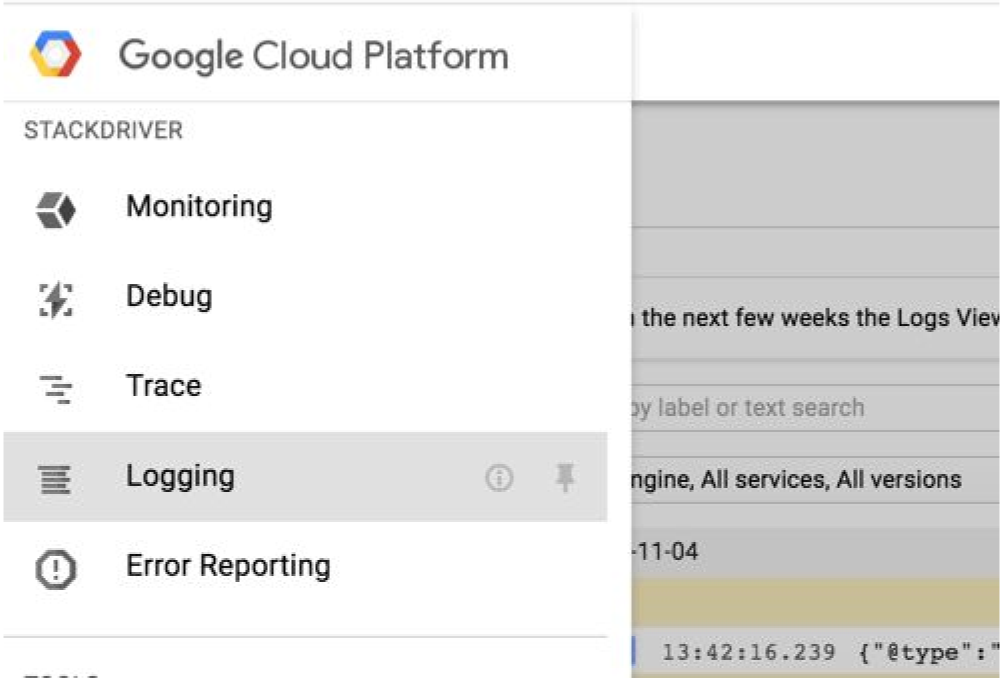
Open a new browser tab/window, and browse to the URL associated with `kubernetes-dashboard`. Enter the username/password to login:



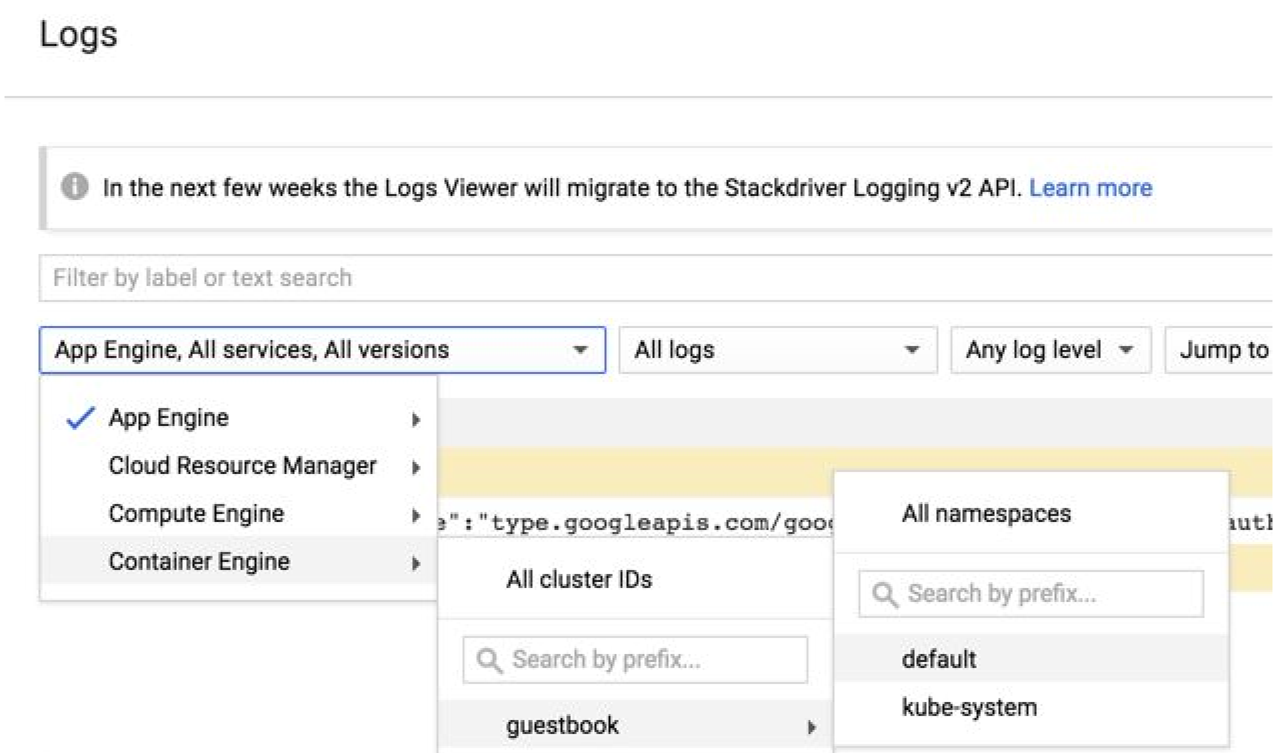

## Google Cloud Logging

Duration: 3:00

During the lab, you've used `kubectl logs` command to retrieve the logs of a container running inside of Kubernetes. When you use Google Container Engine to run managed Kubernetes clusters, all of the logs are automatically forwarded and stored in Google Cloud Logging. You can see all the log output from the pods by navigating to **Stackdriver** > **Logging** in the Google Cloud console:




Once in the logging console, you can navigate to **Container Engine** > **guestbook** > **default** to see all of the logs collected from STDOUT:




From here, you can optionally export the logs into Google BigQuery for further log analysis, or setup [log-based alerting](). We won't get to do this during the lab today.

# Kubernetes - The Advanced

## Manage Compute Resources

You can specify the resource needs for each of the containers within the Deployment descriptor file. By default, each container is given 10% of a CPU and no memory use restrictions. You can see the current resource by describing a Pod instance, look for the Requests/Limits lines.

```
$ kubectl describe pod helloworld-ui-...
Name:           helloworld-ui-3230519151-cusci
Namespace:      default
Node:           gke-guestbook-default-pool-ad6862db-tvu0/10.132.0.4
Start Time:     Mon, 07 Nov 2016 23:51:56 +0100
Labels:         app=helloworld-ui
                pod-template-hash=3230519151
                version=1.0
                visualize=true
Status:         Running
IP:             10.104.1.6
Controllers:    ReplicaSet/helloworld-ui-3230519151
Containers:
  helloworld-ui:
    Container ID:
docker://03c6b58d866c721611ebdabbd243716437119a2b300d513a3c040abf73c26b6d
    Image:              saturnism/spring-boot-helloworld-ui:v1
    Image ID:
docker://sha256:e3757c8390a74528641c71ec43077a00811dd8dca24de9738a678aa51bb6e4fb
    Port:               8080/TCP
    Requests:
      cpu:              100m
    State:              Running
      Started:          Mon, 07 Nov 2016 23:53:01 +0100
```

In Kubernetes, you can reserve capacity by setting the Resource Requests to reserve more CPU and memory:

First, edit the deployment:

```
$ kubectl edit deployment helloworld-ui
```

Add the CPU and memory requests explicitly:

```
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  ...
spec:
  ...
  template:
    ...
    spec:
```

```
      containers:
      - name: helloworld-ui
        image: saturnism/spring-boot-helloworld-ui:v1
        ...
        resources: {} ## Find this & Remove. Replace with the following:
        resources:
          requests:
            cpu: 200m
            memory: 128Mi
        ...
```

In Kubernetes, CPU can be divided into milli-cores (0.1% of a CPU). 1000m means one thousand milli-cores, which is 1 CPU. In this example, 500m means 50% of a CPU.

If the application need to consume more CPU - that's OK as well, the applications are allowed to burst. You can also set an upper limit to how much the application burst by setting the Resource Limit:

First, edit the deployment:

```
$ kubectl edit deployment helloworld-ui
```

Add the CPU and memory limits:

```
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  ...
spec:
  ...
  template:
    ...
    spec:
      containers:
      - name: helloworld-ui
        image: saturnism/spring-boot-helloworld-ui:v1
        resources:
          requests:
            ...
          limits:
            cpu: 500m
            memory: 256Mi
        ...
```

## Health Checks

Duration: 10:00

During rolling update, a pod is removed as soon as a newer version of pod is up and ready to serve. By default, without health checks, Kubernetes will route traffic to the new pod as soon as the pods starts. But, it's most likely that your application will take sometime to start, and if you route traffic to the application that isn't ready to serve, your users (and/or consuming services) will see errors. To avoid this, Kubernetes comes with two types of checks: Liveness Probe, and Readiness Probe.

After a container starts, it is not marked as Healthy until the Liveness Probe succeeds. However, if the number of Liveness Probe failures exceeds a configurable failure threshold, Kubernetes will mark the pod unhealthy and attempt to restart the pod.

When a pod is Healthy doesn't mean it's ready to serve. You may want to warm up requests/cache, and/or transfer state from other instances. You can further mark when the pod is Ready to serve by using a Readiness Probe.

Let's add a Liveness Probe to our Helloworld Service by editing the Deployment:

```
$ kubectl edit deployment helloworld-service --record
```

In the editor, add a Liveness Probe:

```
apiVersion: extensions/v1beta1
kind: Deployment
…
spec:
…
  template:
    …
    spec:
      …
      containers:
      - image: saturnism/spring-boot-helloworld-service:...
        livenessProbe:
          httpGet:
            path: /hello/healthcheck
            port: 8080
      ...
```

**Note:** You can configure both Liveness Probe and Readiness Probe by checking via a HTTP GET request, a HTTPS GET request, TCP port connectivity, or even a shell script! See the Liveness and Readiness Probe Guide more information.

You can add a Readiness Probe in the similar way:

```
$ kubectl edit deployment helloworld-service --record
```

In the editor, add a Readiness Probe:

```
apiVersion: extensions/v1beta1
kind: Deployment
…
```

```
spec:
…
  template:
    …
    spec:
      …
      containers:
      - image: saturnism/spring-boot-helloworld-service:...
        readinessProbe:
          httpGet:
            path: /hello/ready
            port: 8080
      ...
```

**Note:** In a production scenario, the Liveness Probe and the Readiness Probe will probably be different. Your application may be alive, but it's not ready to serve. E.g., you may want to preload cache after startup, but don't serve until the cache is preloaded.

## Graceful Shutdown

Duration: 5:00

When a pod needs to be deleted (such as reducing the number of replicas), Kubernetes will send the SIGTERM signal to the process. The process should perform all the cleanups. However, we cannot wait forever for the process to exit. By default, Kubernetes waits 30 seconds before sending the final SIGKILL signal to kill the process. If your process needs more or less time, you can configure this through termination grace periods configuration (see guide).

Optionally, you can also ask Kubernetes to execute a shutdown command via the pre-stop lifecycle hook. Read through the Lifecycle Hooks and Termination Notice Guide the learn more - we won't implement this during the lab.

## Configuring Your Application

Duration: 20:00

The Helloworld Service is configured to return a message that uses the following template, configured in the `examples/helloworld-service/application.properties` file:

```
greeting=Hello $name from $hostname with $version
```

There are several ways to update this configuration. We'll go through a couple of them, including:

- Environmental variable
- Command line argument
- and, Config Map

### Environmental Variable

Spring applications can read the override configuration directly from an environmental variable. In this case, the environmental variable is defaulted to `GREETING`. You can specify the environmental variable directly in the Deployment as well, by first edit the Deployment:

```
$ kubectl edit deployment helloworld-service --record
```

In the editor, add the environmental variable:

```
apiVersion: extensions/v1beta1
kind: Deployment
…
spec:
…
  template:
    …
    spec:
      …
      containers:
      - image: saturnism/spring-boot-helloworld-service:...
        env:
        - name: GREETING
          value: Hello $name from envvar!
      ...
```

Again, through the use of Deployments, it'll rolling update all the replicas with the new configuration! If you were to refresh the application, you'll notice that there are no intermittent errors because we also have health checks and readiness checks in place.

Check the application to see it is using the new Greeting string.

Let's rollback to the previous state:

```
$ kubectl rollout undo deployment helloworld-service
deployment "helloworld-service" rolled back
```

### Command Line Argument

Next, let's add a configuration via the command line arguments. You know the drill, edit the Deployment, and add the following section:

```
apiVersion: extensions/v1beta1
kind: Deployment
…
spec:
…
  template:
    …
    spec:
      …
      containers:
```

```
      - image: saturnism/spring-boot-helloworld-service:...
        args:
        - --greeting="Hello $name from args"
      ...
```

**Note:** Yes, there are 3 dashes. The first dash is required by YAML to indicate that this is a list element, followed by a space and two more dashes that is actually passed into the command line argument.

Check the application and submit a name and message to see it is using the new greeting string.

Let's rollback to the previous state again:

```
$ kubectl rollout undo deployment helloworld-service
deployment "helloworld-service" rolled back
```

### Using ConfigMap

In this section, we'll use Kubernetes 1.2's new feature, ConfigMap, to configure the application. You can store multiple text-based configuration files inside of a single ConfigMap configuration. In our example, we'll store Spring's `application.properties` into a ConfigMap entry.

First, update the `examples/helloworld-service/application.properties` with a new configuration value:

```
greeting=Hello $name from ConfigMap
```

Next, create a ConfigMap entry with this file:

```
$ cd examples/helloworld-service/application.properties
$ kubectl create configmap greeting-config --from-file=application.properties
configmap "greeting-config" created
```

Let's take a look inside the newly created entry:

```
$ kubectl edit configmap greeting-config
```

You'll see that the `application.properties` is now part of the YAML file:

```
apiVersion: v1
data:
  application.properties: |
    greeting=Hello $name from ConfigMap
kind: ConfigMap
...
```

You can, of course, edit this ConfigMap in the editor too. If you do, edit only the value for the `greeting` variable.

There are several ways to access the values in this ConfigMap:

- Mount the entries (in our case, `application.properties`) as a file.

- Access from the Kubernetes API (we won't cover this today).

Let's see how we can mount the configurations as files under a specific directory, e.g., `/etc/config/application.properties` .

First, edit the Helloworld Service Deployment:

```
$ kubectl edit deployment helloworld-service --record
```

In the editor, add volumes and volume mounts (important - indentation matters!):

```
apiVersion: extensions/v1beta1
kind: Deployment
...
spec:
...
  template:
    ...
    spec:
      volumes:
      - name: config-volume
        configMap:
          name: greeting-config
      containers:
      - image: saturnism/spring-boot-helloworld-service:...
        args:
        - --spring.config.location=/etc/config/application.properties
        volumeMounts:
        - name: config-volume
          mountPath: /etc/config
      ...
```

This will make the configuration file available as the file `/etc/config/application.properties` and tell Spring Boot to use that file for configuration. Let's verify by going into the pod itself (remember how to do this by using `kubectl exec`?):

First, find the pod name:

```
$ kubectl get pods
NAME                                  READY     STATUS    RESTARTS   AGE
...
helloworld-service-2258836722-arv2f   1/1       Running   0          1m
helloworld-service-2258836722-exlll   1/1       Running   0          1m
...
```

Then, run a shell inside the pod, and see what's in `/etc/config`:

```
$ kubectl exec -ti helloworld-service-2258836722-arv2f /bin/bash
root@helloworldservice-...:/app/src# ls /etc/config
application.properties
root@helloworldservice-...:/app/src# cat /etc/config/application.properties
```

```
...
root@helloworldservice-...:/app/src# exit
```

**Note:** Don't forget to exit out of the pod environment!

Check the application to see it is using the new greeting string.

Last, but not least, you can also specify simple key/value pairs in ConfigMap, and then expose them directly as environmental variables too. See the ConfigMap guide for more examples.

## Managing Credentials

Duration: 20:00

ConfigMap is great to store text-based configurations. Depending on your use cases, it may not be the best place to store your credentials (which sometimes may be a binary file rather than text). Secrets can be used to hold sensitive information, such as passwords, OAuth tokens, and SSH keys. Entries in Secrets are Base64 encoded. However, Secrets are not additionally encrypted when stored in Kubernetes.

In this section, we'll create a Secret that contains the MySQL username and password. We'll subsequently update both the MySQL Replication Controller and the Guestbook Service to refer to the same credentials.

First, let's create a Secret with username and password the command line:

```
$ kubectl create secret generic mysql-secret \
      --from-literal=username=root,password=yourpassword
secret "mysql-secret" created
```

If you look into the newly created Secret, you'll see that the values are Base64 encoded:

```
$ kubectl edit secret mysql-secret
```

In the Editor, you'll see:

```
apiVersion: v1
data:
  password: eW91cnBhc3N3b3Jk
  username: YXBw
kind: Secret
...
```

In the pods, you can access these values a couple of ways:

- Mount each entry as a file under a directory (similar to what we did with ConfigMap)
- Use Downward API to expose each entry as an Environmental Variable (which you can also do with ConfigMap).

Next, configure the Guestbook Service, by editing the Deployment and updating the Environmental Variables too:

```
$ kubectl edit deployment guestbook-service
```

Then, add a couple of Environmental Variables:

```
apiVersion: extensions/v1beta1
kind: Deployment
…
spec:
  …
  template:
  …
    spec:
      …
      containers:
      - image: saturnism/guestbook-service:latest
        env:
        - name: SPRING_DATASOURCE_USERNAME
          valueFrom:
            secretKeyRef:
              name: mysql-secret
              key: username
        - name: SPRING_DATASOURCE_PASSWORD
          valueFrom:
            secretKeyRef:
              name: mysql-secret
              key: password
...
```

And that's it!

Once the deployment completes, check that the application is still working..

## Autoscaling

Duration: 5:00

Kubernetes 1.2 has built-in Horizontal Pod Autoscaling based on CPU utilization (and custom metrics!). We will only cover autoscaling based on CPU utilization in this lab, since the custom metrics scaling is still in Alpha.

To set up horizontal autoscaling is extremely simple:

```
$ kubectl autoscale deployment helloworld-service --min=2 --max=10 --cpu-percent=80
```

Behind the scenes, Kubernetes will periodically (by default, every 30 seconds) collect CPU utilization and determine the number of pods needed.

You can see the current status of the autoscaler by using the describe command:

```
$ kubectl describe hpa helloworld-service
Name:                           helloworldservice
```

```
Namespace:                      default
Labels:                         <none>
Annotations:                    <none>
CreationTimestamp:              Tue, 19 Apr 2016 03:02:18 +0200
Reference:                      Deployment/helloworldservice/scale
Target CPU utilization:         80%
Current CPU utilization:        21%
Min replicas:                   2
Max replicas:                   10
```

It's going to be a little difficult to generate the load needed to kick off the autoscaler. If you are interested, try to install and use Apache HTTP server benchmarking tool `ab`. We won't do that during the lab.

Learn more about Horizontal Pod Autoscaling in the [guide](guide).

## Running Daemons on Every Machine (Optional)

Duration: 5:00

When running pods using Replication Controllers, Replica Set, or Deployments, the pods can be scheduled to run on any machines in the Kubernetes cluster, and sometimes, the more than 1 of pod of the same application could be running on the same machine. This is not a behavior you'd want if you want to have exactly one instance of the pod on each of the machines in the Kubernetes clusters.

A Daemon Set ensures that all (or some) nodes run a an instance of the pod. As nodes are added to the cluster, pods are added to them. As nodes are removed from the cluster, those pods are garbage collected. This is great for running per-machine daemons, such as:

- cluster storage daemon on every node, such as `glusterd`, `ceph`, etc.
- logs collection daemon on every node, such as `fluentd` or `logstash`, etc.
- monitoring daemon on every node, such as Prometheus Node Exporter, `collectd`, etc.

For this lab, we'll deploy a Cassandra cluster as a Daemon Set because we want to ensure exactly one Cassandra node is running on every machine. Luckily, there is an example of this on GitHub: https://raw.githubusercontent.com/kubernetes/kubernetes/master/examples/storage/cassandra/cassandra-daemonset.yaml. Examine the file carefully - it's almost the same as a Deployment descriptor, but with the kind as DaemonSet.

You can actually deploy this Daemon Set descriptor from GitHub directly:

```
$ kubectl create -f  \
https://raw.githubusercontent.com/kubernetes/kubernetes/master/examples/storage/cassand
ra/cassandra-daemonset.yaml
```

The takeway from this YAML file is that the kind is a `DaemonSet` rather than a Deployment.

There is a chance that the node we have doesn't have enough capacity to run this instances. You may need to delete a few existing deployments, or resize the cluster, before the the node has capacity to run this. You can verify that there is one instance running on every machine:

```
$ kubectl get pods -owide
NAME                                    READY     STATUS    RESTARTS   AGE       NODE
cassandra-0oy09                         1/1       Running   0          4m
gke-guestbook-default-pool-622f144b-6rrv
cassandra-7nrcq                         1/1       Running   0          4m
gke-guestbook-default-pool-622f144b-be8d
cassandra-kw2kc                         1/1       Running   0          4m
gke-guestbook-default-pool-622f144b-56tt
cassandra-sfoep                         1/1       Running   0          4m
gke-guestbook-default-pool-622f144b-bpfa
```

You can expose all of these Pods via a Service exactly the same way that you expose pods with Replication Controllers, Replica Set, or Deployments. At the end of the day, Service will route traffic based on label selectors, so it doesn't matter how a Pod was initially created.

There is a Service descriptor on GitHub you can deploy as well:
https://raw.githubusercontent.com/kubernetes/kubernetes/master/examples/storage/cassandra/cassandra-service.yaml
(hint: use `kubectl create -f <url to the raw file>` ).

You can delete all of the pod instances by deleting the Daemon Set:

```
$ kubectl delete daemonset cassandra
```

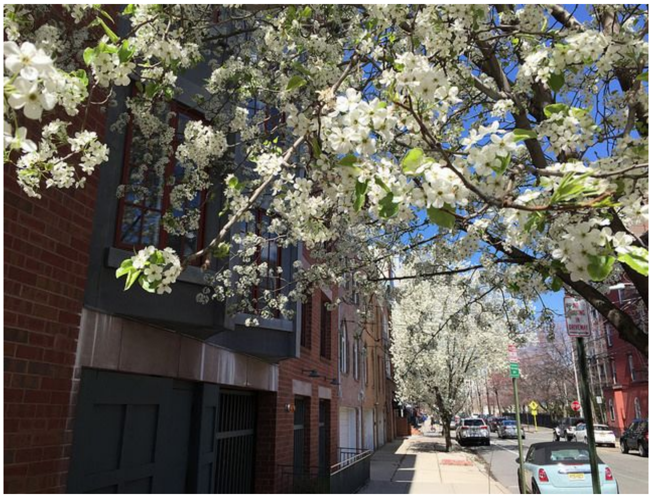
## Managing Batched / Run-Once Jobs

Duration: 15:00

So far, the lab has been showing how to run long running serving processes. What if you need to run a one-time job, such as a batch process, or simply leveraging the cluster to compute a result (like computing digits of Pi)? You shouldn't use Replication Controllers, Replica Set, or Deployments to run a job that is expected exit once it completes the computation (otherwise, upon exit, it'll be restarted again!).

Kubernetes supports running these run-once jobs, which it'll create one or more pods and ensures that a specified number of them successfully terminate. When a specified number of successful completions is reached, the job itself is complete. In many cases, you'll have run a job that only need to complete once.

For this lab, we'll run a job that uses [Google](#) [DeepDream](#) to produce a dreamy picture. The job will retrieve an image from the web, processes it with Google DeepDream, and then output the processed image into a Google Cloud Storage bucket, like this:




First, create a Google Cloud Storage bucket that will be used to store the image. Bucket names are globally unique. Use the Project ID as the bucket name to minimize conflicts with other bucket names:

```
$ gsutil mb gs://<project_id>
```

Next, find an interesting image to process. Since the processing time will increase with the size of the image, please find a JPG image that's no larger than 640px by 480px large. Make sure you grab the public URL to the image, e.g., this one: https://farm2.staticflickr.com/1483/25947843790_7cf8d5e59c_z_d.jpg

Then, you can launch a job directly from the command line using `kubectl run` with the `--restart=Never` argument. This tells Kubernetes that this is a Run Once job:

```
$ kubectl run deepdream-1 --restart=OnFailure \
      --image=saturnism/deepdream-cli-gcs -- \
      -i 1 --source=<an image url> \
      --bucket=<project_id> --dest=output.jpg
```

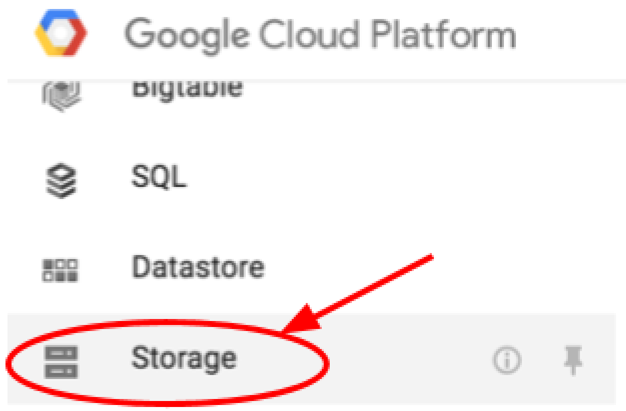
**Note:** You can also create a Job descriptor as well. Read the Kubernetes Jobs Guide to learn how you can write a YAML file to schedule a job.

This job should finish relatively quickly. You'll be able to see the status of the job via the command line:

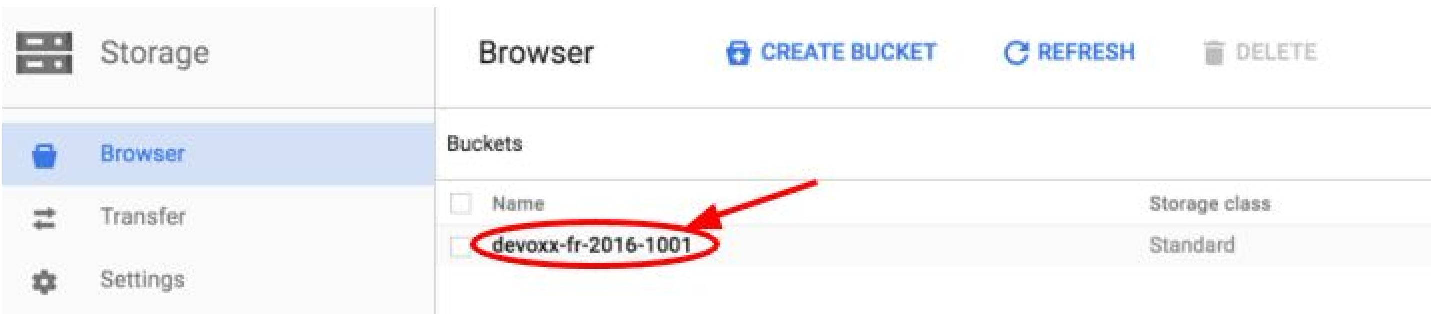
```
$ kubectl get jobs
NAME            DESIRED   SUCCESSFUL   AGE
deepdream-1     1         0            13m
```

Wait until the successful count is 1. Once the job finishes, browse to your Google Cloud Storage bucket and check the output.

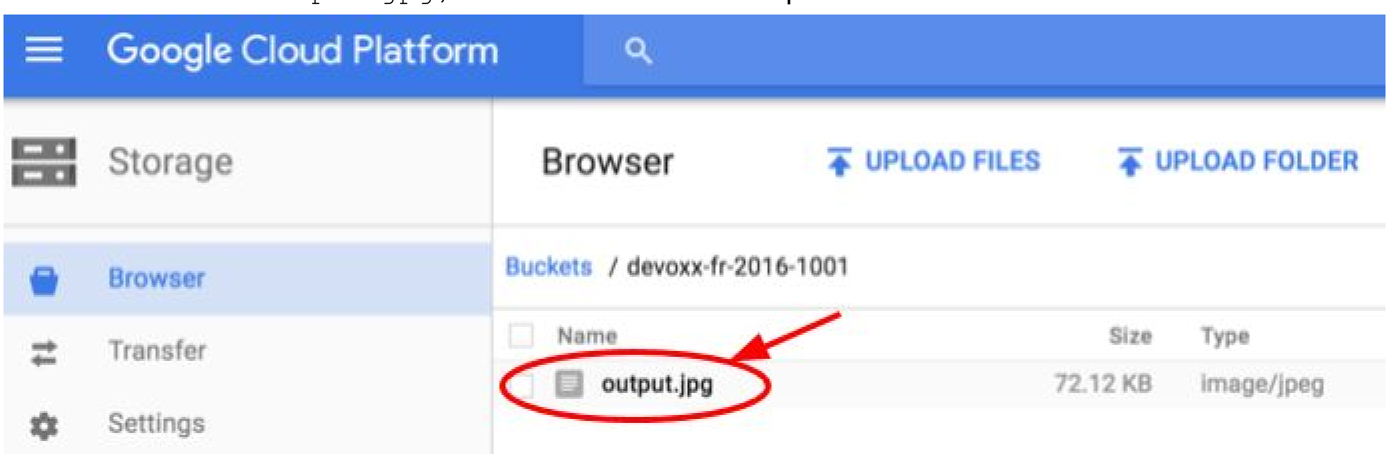
First, navigate to **Storage** > **Storage**:




Then, navigate to the bucket (your Project ID is the bucket name):




You should see the `output.jpg`, click on it and see the output.




This first iteration is probably not very impressive due to the configuration parameter we used.

Now that we know this job works, let's run a longer job:

```
$ kubectl run deepdream-2 --restart=OnFailure \
      --image=saturnism/deepdream-cli-gcs -- \
      -o 6 -l conv2/3x3 --source=<an image url> \
      --bucket=<project_id> --dest=output-2.jpg
```

This job will take a couple of minutes to complete. While it's running, you can see the pod that was started, and also tail its logs (recall how you can do that via the command line).

Regardless of whether the job is running or already finished, you can retrieve the job output by first, getting the Pod name associated with the job:

```
$ kubectl get pods --selector=job-name=deepdream-2 --show-all
NAME                READY     STATUS      RESTARTS   AGE
deepdream-2-...     0/1       Completed   0          1m
```

Then, you can use `kubectl logs` to retrieve the job output:

```
$ kubectl logs deepdream-2-czsrk
libdc1394 error: Failed to initialize libdc1394
WARNING: Logging before InitGoogleLogging() is written to STDERR
I1108 00:06:20.109063     1 net.cpp:42] Initializing net from parameters:
name: "GoogleNet"
input: "data"
input_dim: 10
input_dim: 3
input_dim: 224
input_dim: 224
force_backward: true
...
```

Try calculating Pi to the 2,000 place in a Job and use the above technique to find its value!

```
$ kubectl run pi --image=perl --restart=OnFailure -- perl -Mbignum=bpi -wle 'print
bpi(2000)'
```

Last, but not least, even though we launched the jobs from the command lines, you can always write a Job descriptor as a YAML or JSON file and submit those descriptors as well.

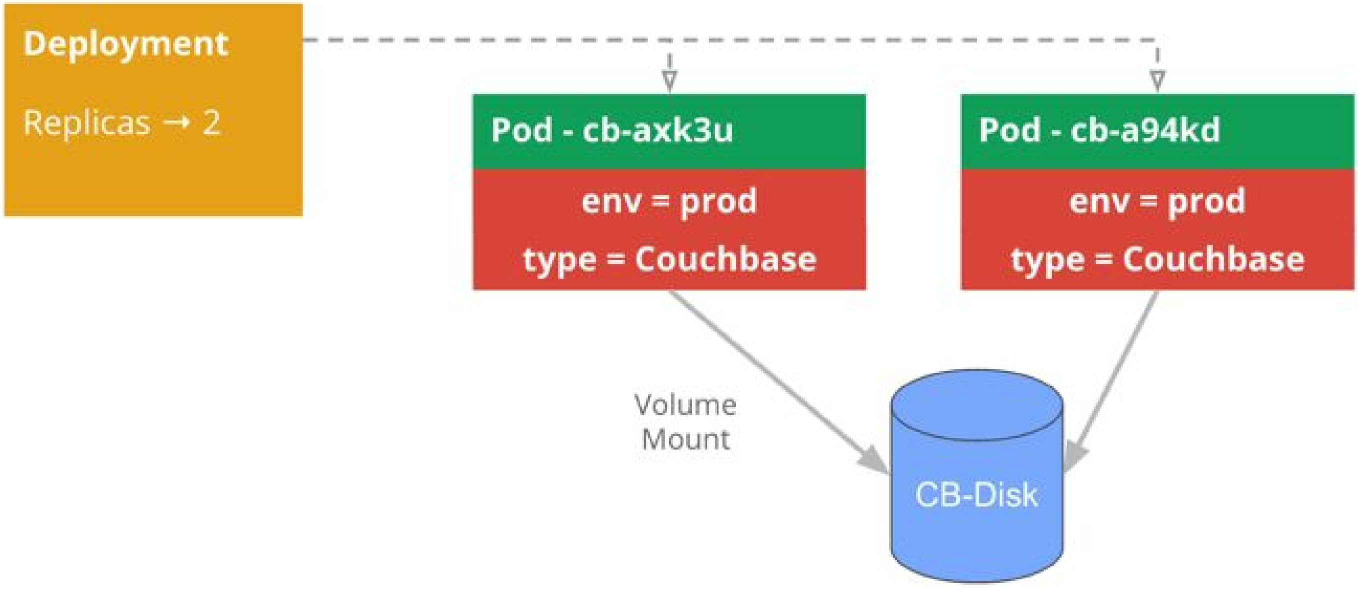
> **Note:** Kubernetes 1.4 has Alpha support for Scheduled Jobs. Thinks of it as a distributed crontab. You can learn more in [Kubernetes Scheduled Job Guide](Kubernetes Scheduled Job Guide).

## StatefulSets

In this section, we'll deploy Couchbase to Kubernetes using Arun Gupta's couchbase example that Ray Tsang collaborated on. The example is here: https://github.com/arun-gupta/couchbase-kubernetes, but you don't need to clone this repository.

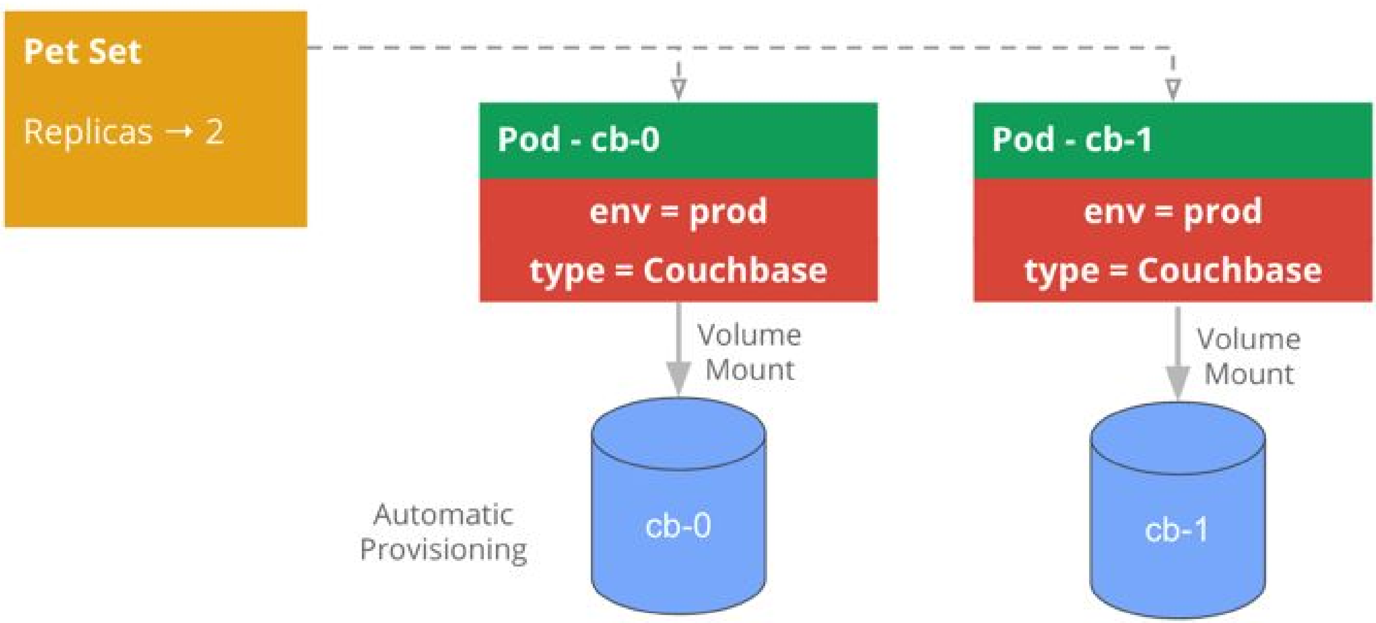
If each of your application instance is stateful (i.e., stores instance-specific data), you'll run into issues quickly if you use a normal Kubernetes Deployment, because each of the replica will be pointing to exactly the same persistent volume for storage - and it may not work.

Take Couchbase for example - each Couchbase node will need to store its own sharded data. If you used a regular Deployment, it'll look like this:




Not only that, each Pod will have a ephemeral Pod name, making it impossible to tell which Pod is the primary/master/first.

With StatefulSet, each Pod can have its own persistent volumes - and the names become stable, i.e., the first instance of the StatefulSet will have the ID of 0, and second instance will have ID of 1:




StatefulSet is a beta a feature in Kubernetes 1.5, which is automatically enabled in Google Container Engine 1.5.x.

Using a headless service is important in StatefulSet. It will provide stable DNS names such as `couchbase-0` for the first instance, and `couchbase-1` for the second instance. Instance name is important in this example deployment, because all `couchbase-0` will act as the master instance. Let's provision the service:

```
$ kubectl apply -f \
https://raw.githubusercontent.com/arun-gupta/couchbase-kubernetes/master/cluster-statef
ulset/couchbase-service.yml
```

In this YAML file, notice of a couple of important details:

- The first service has a `ClusterIP` attribute to to `None`. This creates a headless service. Unlike the regular service which provides a stable IP address acting as a network load balancer, a headless service doesn't create a stable IP address. Instead, in this case, it allows the creation of stable Pod hostnames, such as `couchbase-0`, `couchbase-1`, and so on.
- The second service has a `sessionAffinity` attribute set to `ClientIP`. This make sure your browser is connection is pinned to one of the many instances based on your IP address.

Finally, we can deploy Couchbase as a StatefulSet:

```
$ kubectl apply -f \
https://raw.githubusercontent.com/arun-gupta/couchbase-kubernetes/master/cluster-statef
ulset/couchbase-statefulset.yml
```

In this YAML file, notice of a couple of important details:

- The kind is a StatefulSet rather than a Deployment or ReplicaSet
- You can depend on `couchbase-0` being provisioned before `couchbase-1`. Provision of the instances sequential, from the first instance to to the number of replicas you need.
- `volumeClaimTemplates` is used to automatically generate a new Persistent Volume Claim, and subsequently, this will automatically provision a disk in Google Cloud Platform with the specified capacity. We didn't need to create a Google Compute Engine disk manually.

You can scale the Statefulset Set just like a Deployment:

```
$ kubectl scale statefulset couchbase --replicas=3
```

This will create the a Pod with stable name of `couchbase-2`, and also automatically provision a new disk. You can see all of the disks that were automatically created:

```
$ kubectl get pv
NAME      CAPACITY   ACCESSMODES   RECLAIMPOLICY   STATUS    CLAIM
REASON    AGE
pvc-...   1Gi        RWO           Delete          Bound
default/couchbase-data-couchbase-0             10m
pvc-...   1Gi        RWO           Delete          Bound
default/couchbase-data-couchbase-1             10m
pvc-...   1Gi        RWO           Delete          Bound
default/couchbase-data-couchbase-2             38s
```

You can login into the Couchbase UI by first finding the couchbase-ui external IP address:

```
$ kubectl get svc
NAME           CLUSTER-IP   EXTERNAL-IP     PORT(S)    AGE
couchbase      None         <none>          8091/TCP   12m
couchbase-ui   10.3.250.7   IP_ADDRESS      8091/TCP   12m
kubernetes     10.3.240.1   <none>          443/TCP    13m
```

Then, open the browser and navigate to `http://MY_IP_ADDRESS:8091`
The username is `Administrator` and password is `password`.

You can delete all of the Pod instances by deleting the StatefulSet:

```
$ kubectl delete statefulset couchbase
```

### Other Goodies

There are many more features that are not covered in this lab. It's good to know those features exists though. For example:

- Ingress - Provision HTTP/HTTPs Load Balancer.
- Namespaces - Create virtual clusters, so that you can host multiple teams, projects, or environments.
- StatefulSet - Provision stateful applications that require stable names and instance-specific storage (e.g., running Couchbase, Zookeeper, etc). Storage can be dynamically provisioned.
- Third-party Resource - You can create your own resource descriptors and provision your own controllers!
- Federation - You can federate multiple clusters together and control it through a single control plane.
- Security features: Role-Based Access Control, Network Policy and Pod Security Policy

## Cleanup : Shut down your cluster!!!!

Duration: 5:00

Don't forget to shut down your cluster, otherwise they'll keep running and accruing costs. The following commands will delete the persistent disk, the GKE cluster, and also the contents of the private repository.

```
$ gcloud container clusters delete guestbook
$ gcloud container clusters delete couchbase
$ gcloud compute disks delete mysql-disk

$ gsutil ls
gs://artifacts.<PROJECT_ID>.appspot.com/
...

$ gsutil rm -r gs://artifacts.<PROJECT_ID>.appspot.com/
Removing gs://artifacts.<PROJECT_ID>.appspot.com/...

$ gsutil rm -r gs://<PROJECT_ID>/
Removing gs://<PROJECT_ID>/...
```

Of course, you can also delete the entire project but note that you must first disable billing on the project. Additionally, deleting a project will only happen after the current billing cycle ends.

## Extra Credit

Duration: 10:00

Here are some ideas for next steps.

### Run Kubernetes Locally with Minikube

You can run Kubernetes locally with Minikube. See [https://github.com/kubernetes/minikube](https://github.com/kubernetes/minikube) for instructions.

### Install Cloud SDK Command Line tool locally

To use `gcloud` command line locally, you'll need to install Cloud SDK. Follow the Cloud SDK installation guide for your platform.

### Create a Docker Machine on Google Compute Engine

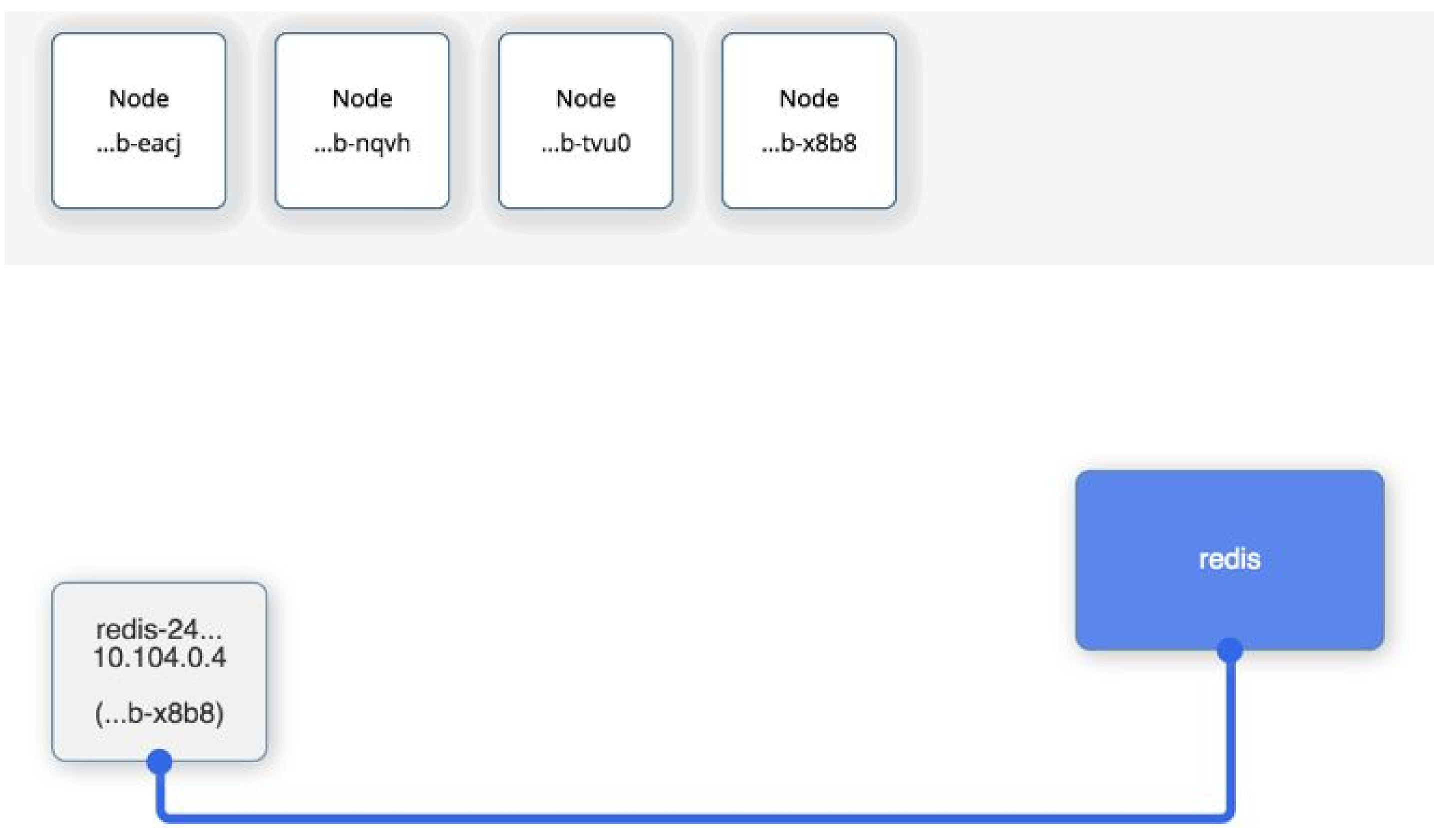
You can create a Docker Machine on Google Compute Engine rather than Virtualbox. You can see some of neat tips and tricks on Ray's blog on My Slow Internet vs Docker.

### DIY Kubernetes cluster on Compute Engine

Download the open source version, build it and deploy a cluster yourself with the kubernetes tools. Check out the Kubernetes Getting Started documentation. This can be as simple as running: '`curl -sS https://get.k8s.io | bash`'

### Deploy a Visualizer

There is a nice visualizer tool that was originally created by Brendan Burns, a Kubernetes engineer. There is a fork of the visualizer that can display nodes and more under the GitHub repository: [https://github.com/saturnism/gcp-live-k8s-visualizer](https://github.com/saturnism/gcp-live-k8s-visualizer) that looks like this:



**Warning:** The visualizer will continuously poll the current states via the proxy. It will increase internet usage, decrease laptop battery life, and in some cases, increase CPU usage.

If you find the browser to slow down, refresh the visualizer page, or close it :)

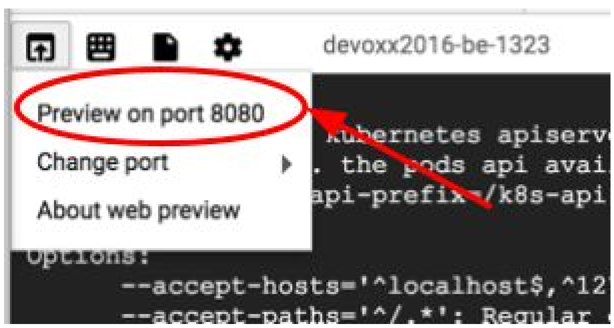
You can install the visualizer for this lab if you like - but it'll take a little time and extra steps. First, clone the GitHub repository, and let's go into it:

```
$ git clone https://github.com/saturnism/gcp-live-k8s-visualizer
$ cd gcp-live-k8s-visualizer
$ git checkout kubernetes-1.2
```

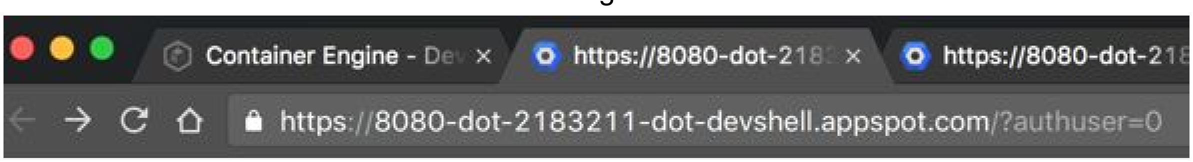
You can then start a local Kubernetes proxy that will forward requests from the proxy to the Kubernetes server via a secure proxy. It's important to know that you can issue any Kubernetes API calls to the proxy without additional authentication:

```
$ kubectl proxy --address=0.0.0.0 -w . -p 8080
Starting to serve on [::]:8080
```

Finally, use Web Preview to see the visualizer from the Cloud Shell environment:




You will first see an Unauthorized error message:

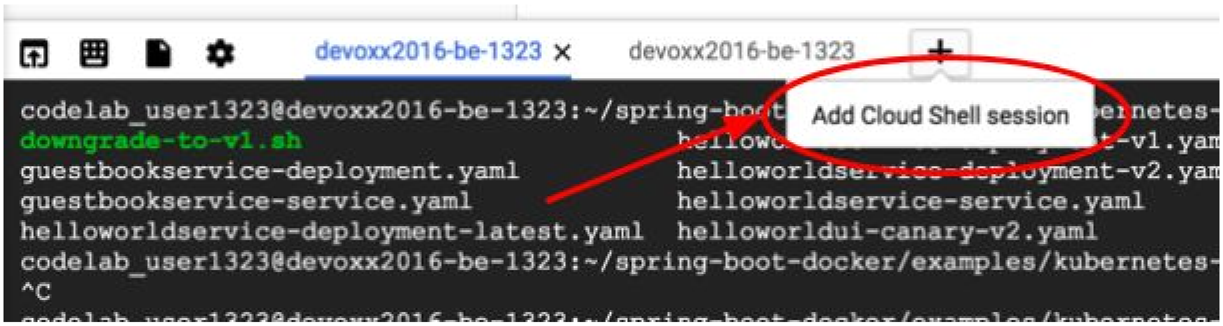



<h3>Unauthorized</h3>

In the URL, remove the query parameter `?authuser=0` and append the path `static/` so that your URL looks like: `https://8080-dot-.....-devshell.appspot.com/static`

That should be it! For the rest of the lab, you'll be able to visualize the changes you are making.

To do the rest of the lab, please open a new Cloud Shell tab by clicking on **+**:




In the newly opened shell, set the default zone and region again:

```
$ gcloud config set compute/zone europe-west1-c
$ gcloud config set compute/region europe-west1
```

# What's next?

Duration: 5:00

**Codelab feedback**

- The codelab was easy and useful
- The codelab was too complicated
- The codelab didn't go far enough
- I had some technical difficulties (please share details using the feedback link)

### Kubernetes

- http://kubernetes.io
- https://github.com/googlecloudplatform/kubernetes
- mailing list: google-containers
- twitter: @kubernetesio
- IRC: #google-containers on freenode

### Minikube

- https://github.com/kubernetes/minikube

### Google Container Engine

- https://cloud.google.com/container-engine/

### Google Compute Engine

- https://cloud.google.com/compute-engine/

## Other Adaptations

Following are a few labs that are created based on this lab:

- Google Compute Engine and Container Engine Lab
- RedHat Developer Kubernetes Lab