

Kubernetes (on Kubernetes Engine) - Basics to Advanced

Self-link: bit.ly/k8s-lab

Author: Ray Tsang [@saturnism](https://twitter.com/saturnism)

Feedback: saturnism.me/feedback

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 an 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 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 but you should feel free to use your favorite environment instead.

Once you are familiar with Kubernetes, there are other more advanced labs you can try:

1. [Istio Lab](https://bit.ly/istio-lab) - bit.ly/istio-lab
2. [Google Cloud Native with Spring Boot, App Engine and Kubernetes](https://bit.ly/spring-gcp-lab) - bit.ly/spring-gcp-lab
3. [Serverless with Spring Cloud Function, Riff, and Knative](https://bit.ly/spring-riff-lab) - bit.ly/spring-riff-lab
4. [Knative](https://bit.ly/knativeworkshop) - bit.ly/knativeworkshop
5. [gRPC Microservices with Java](https://bit.ly/grpc-java-lab-doc) - bit.ly/grpc-java-lab-doc

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

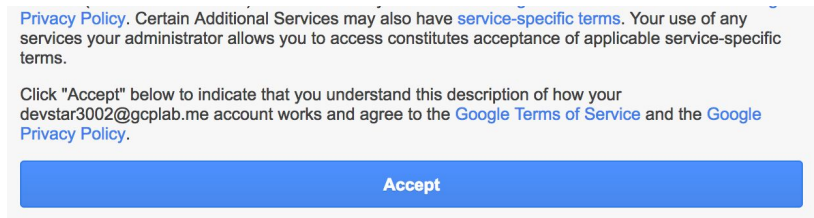
Duration: 5:00

Logging In

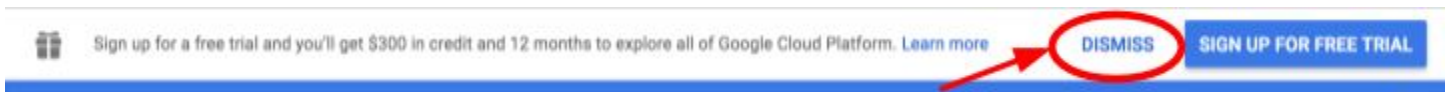
The instructor will provide you with a temporary username / password to login into Google Cloud Console.

Important: To avoid conflicts with your personal account, please open a new incognito window for the rest of this lab.

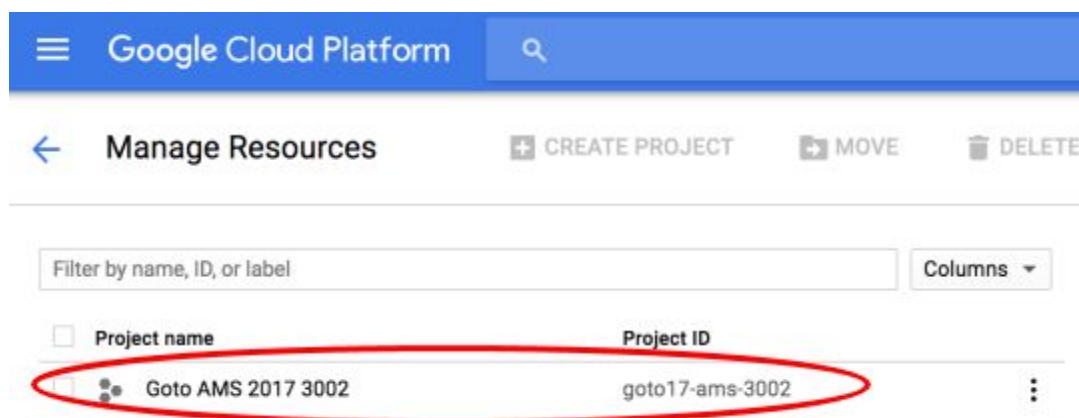
Login into the Cloud Console: <https://console.cloud.google.com/> with the provided credentials. In **Welcome to your new account** dialog, click **Accept**



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.



You should see just one Project - click into it.



When prompted **Update to Terms of Service**, select **Yes** and click **Accept**.

Updates to Terms of Service

We have updated some of our Terms of Service. To continue, accept all the updated Terms of Service below.

I agree that my use of any [services and related APIs](#) is subject to my compliance with the applicable [Terms of Service](#).

☒ Yes ☐ No

[ACCEPT](#)

In the **IAM & Admin** page, wait for permissions to initialize (sometimes, refreshing the page speeds up the process :)

IAM

- Quotas
- Service accounts
- Labels
- GCP Privacy & Security
- Settings
- Encryption Keys
- Identity-Aware Proxy

Permissions for project "Goto AMS 2017 3002"

These permissions affect the entire "Goto AMS 2017 3002" project and all of its resources. To grant permissions, add a member and then select a role for them. Members can be people, domains, groups, or service accounts.

Some roles are in beta development and might be changed or deprecated in the future. [Learn more](#).

Filter by name or role View by: **Members**

Type	Members	Role(s)
<input type="checkbox"/>	devstar3002@gcplab.me	Multiple
<input type="checkbox"/>	projectsowner@gcplab.me	Owner

Cloud Shell

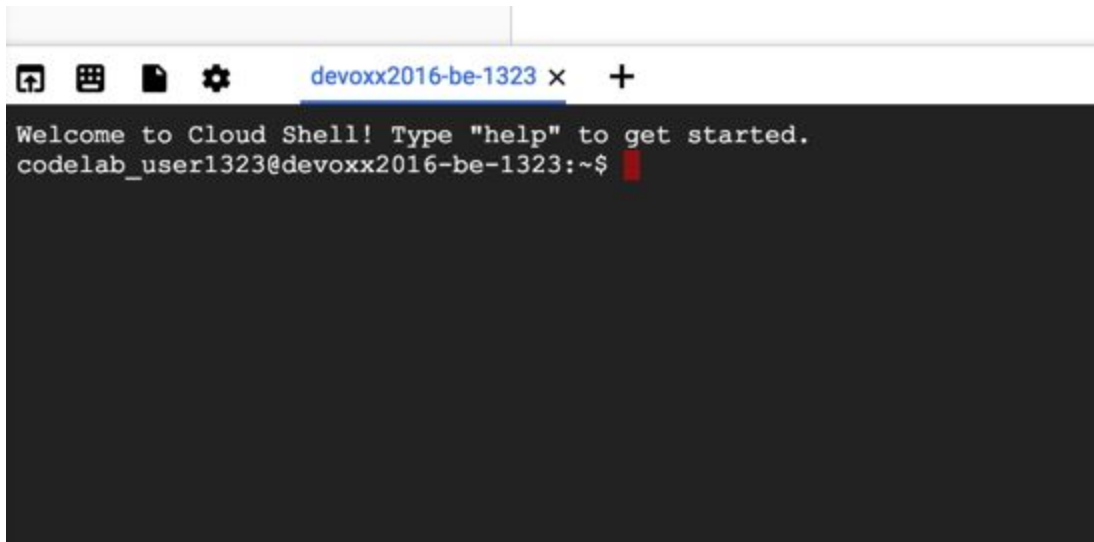
You will do most of the work from the [Google Cloud Shell](#), a command line environment running in the Cloud. This Debian-based virtual machine is loaded with all the development tools you'll need (docker, gcloud, kubectl and others) and offers a persistent 5GB home directory. Open the Google Cloud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



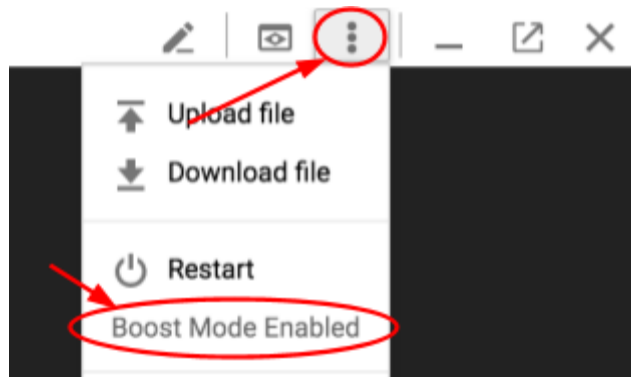
When prompted, click **Start Cloud Shell**:



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



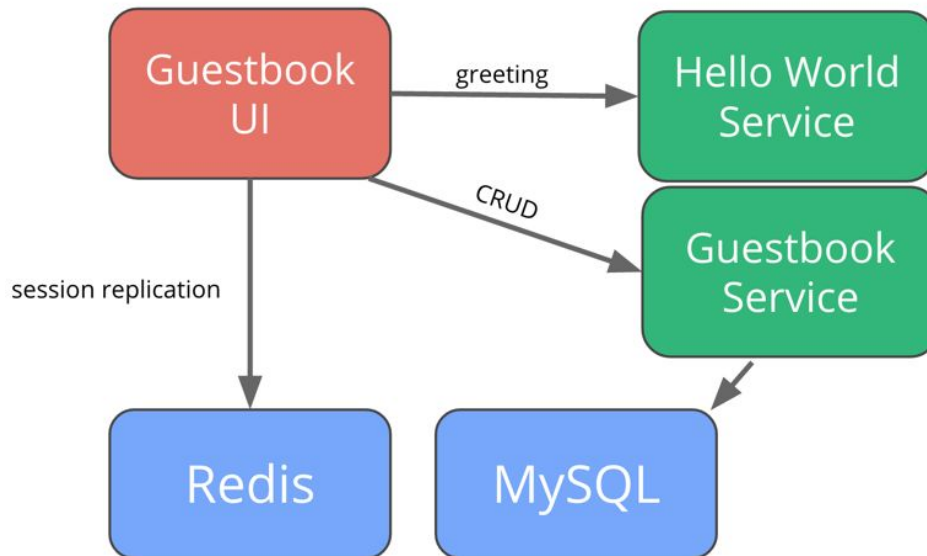
Next, enable **Boost Mode** for Cloud Shell. This will provision a larger VM instance for the shell. We'll need this in order to build and push Docker container faster:



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 Kubernetes Engine.

Everything on GCP is operated with API (even from the console!). Enable the Google Cloud Platform APIs so that you can create a Kubernetes cluster.

```
$ gcloud services enable compute.googleapis.com \
  container.googleapis.com \
  containerregistry.googleapis.com
```

Set default region and zones:

```
$ gcloud config set compute/zone us-east1-d
$ gcloud config set compute/region us-east1
```

Note: For the lab, use the region/zone recommended by the instructor. Learn more about different zones and regions in [Regions & Zones documentation](#).

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

```
$ gcloud container clusters create guestbook \
  --cluster-version 1.13 \
  --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 Kubernetes 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 **Compute Engine** → **VM Instances** page.

Get the Guestbook source

Duration: 3:00

Start by cloning the GitHub repository for the Guestbook application:

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

And move into the kubernetes examples directory.

```
$ cd ~/spring-boot-docker/examples/kubernetes-1.10
```

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.

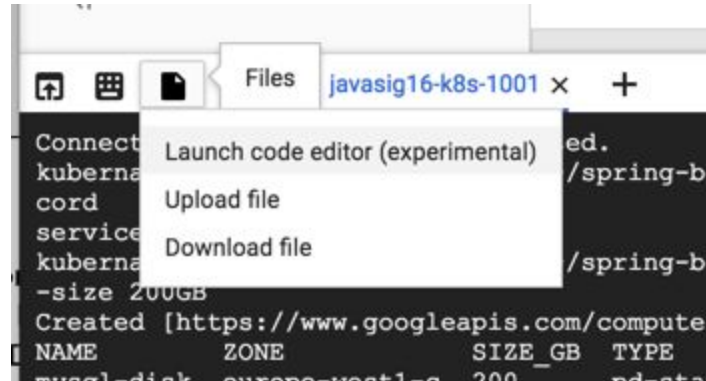
You can start a single Pod in Kubernetes by creating a Pod resource. However, a Pod created this way would be known as a Naked Pod. If a Naked Pod dies/exits, it will not be restarted by Kubernetes. A better way to start a pod, is by using a higher-level construct such as a Deployment.

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. It does this using an object called a ReplicaSet under the covers. You can use deployments to easily:

- Create a Deployment to bring up a ReplicaSet 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.

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 choose to use 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](#). The specification can be found in [Kubernetes API reference](#).

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

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

You should see a Redis instance running:

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

Note down the Pod name, you can kill this Redis instance::

```
$ kubectl delete pod redis-...
pod "redis-..." deleted
```

Kubernetes will automatically restart this pod for you:

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

Kubernetes is container format agnostic. In your lab, we are working with Docker containers. Keep in mind that Kubernetes works 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
055d4b03a1a5	gcr.io/...	"/pause"	5 minutes ago	Up
...				

There are other containers running too. The interesting one is the pause container. The atomic unit Kubernetes can manage is actually a Pod, not a container. A Pod can be composed of multiple tightly-coupled containers that is guaranteed to be scheduled onto the same node, and will share the same Pod IP address, and can mount the same volumes.. What that essentially means is that if you run multiple containers in the same Pod, they will share the same namespaces.

A pause container is how Kubernetes use Docker containers to create shared namespaces so that the actual application containers within the same Pod can share resources.

Important: Make sure you exit from the SSH before you continue.

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

You can also launch a command or start a shell directly within the container using `kubectl exec` command. Let's open a shell inside of the Redis container:

```
$ kubectl exec -ti redis-... /bin/bash
root@redis...:/data# ls /
bin boot data dev etc home ...
```

The Pod name is automatically assigned as the hostname of the container:

```
root@redis...:/data# hostname
redis-....
root@helloworld-...:/app/src# hostname -i
10.104.1.5
root@helloworld-...:/app/src# exit
```

Each Pod is also automatically assigned an ephemeral internal IP address:

```
root@helloworld-...:/data# hostname -i
10.104.1.5
```

Important: make sure you exit from the container shell before you continue.

```
root@helloworld-...:/data# exit
```

Lastly, any of the container output to STDOUT and STDERR will be accessible as Kubernetes logs:

```
$ kubectl logs redis-...
```

You can tail the logs too:

```
$ kubectl logs -f redis-...
```

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:

```
apiVersion: v1
kind: Service
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 --zone=us-east1-d
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: apps/v1
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.

Volume provisioning 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-1.10-simple>.

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.

First, deploy the Hello World:

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

Then, deploy the Guestbook Service Deployment and service too!

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

Once created, you can see the replicas with:

```
$ kubectl get deployment
```

NAME	DESIRED	CURRENT	UP-TO-DATE	AVAILABLE	AGE
<i>helloworld-service</i>	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
<i>helloworld-service-1726325642-fva3a</i>	1/1	<i>Running</i>	0	1m
<i>helloworld-service-1726325642-ujq2o</i>	1/1	<i>Running</i>	0	1m
mysql-3871635011-82u7v	1/1	Running	0	2m
redis-2107737895-mnrx6	1/1	Running	0	11m

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. Find a helloworld-service pod, and delete it. Let's try it!

A word on networking

Duration: 7:00

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 guestbook-service... /bin/bash
root@guestbookservice:/# 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@guestbookservice:/# exit
```

Deploy the Frontend

Duration: 5:00

You know the drill by now. We first need to create the deployment 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 Deployment and the service at the same time:

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

You can find the public IP in the output in a minute or two:

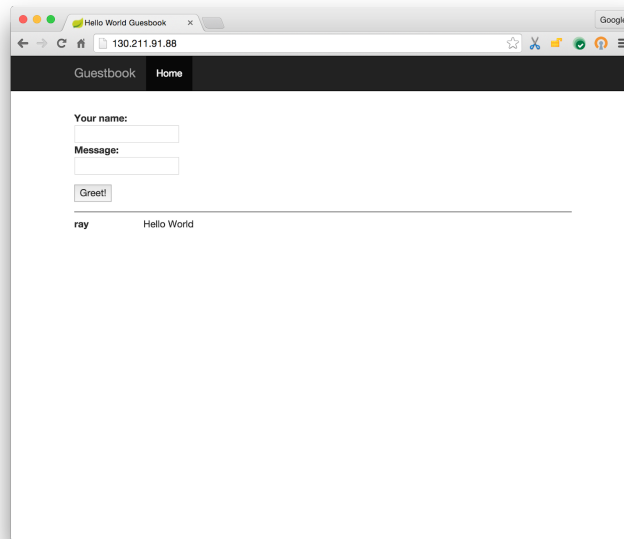
```
$ kubectl get services helloworld-ui
```

NAME	TYPE	CLUSTER-IP	EXTERNAL-IP	PORT(S)	...
helloworld-ui	LoadBalancer	10.47.243.24	X.X.X.X	80:31624/TCP	9m

Note: The external load balancer may take a minute or two to create. Please retry the command above until the `EXTERNAL-IP` entry shows up.

You can now access the guestbook via the External IP address by navigating the browser to `http://EXTERNAL-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 deployment 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=15
```

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

```

Note: At this point, you may be wondering how did we specify how much CPU/memory each pod should use? How do we know how much CPU/memory each node can support? We'll get into this in the later section of the lab!

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 --num-nodes=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 2 so that we can free up resources for the later labs:

```

$ kubectl scale deployment helloworld-service --replicas=2

```

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](#).

First, look up your Project ID:

```
# Find the Project ID
$ gcloud config get-value core/project
Your active configuration is: [cloudshell-XXXX]
yourproject-XXXX ← this is your Project ID!
```

Change into the Helloworld UI source directory:

```
$ cd ~/spring-boot-docker/examples/helloworld-ui
```

Make some changes to `templates/index.html`. Changing the background is always a fun one to try. Once you've made the changes, build and push a new version of the Docker container:

Set the Project ID as an environmental variable. You need the `PROJECT_ID` to create the container image and pushing it to the private registry:

```
$ export PROJECT_ID=$(gcloud config get-value core/project)
$ echo ${PROJECT_ID}
yourproject-XXXX
```

```
# Build the container image in the Docker daemon
$ docker build -t gcr.io/${PROJECT_ID}/helloworld-ui:v2 . # make sure include
                                                         # the . in the end.

# Make sure Docker can authenticate and push to GCR
# This adds auth entries into $HOME/.docker/config.json
$ gcloud auth configure-docker

# Push the image to GCR
$ docker push gcr.io/${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 watch another video? This one about Google Container Registry:
<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, make a copy of the deployment file and changing the content to deploy to the new version

```
$ cd ~/spring-boot-docker/examples/kubernetes-1.10
```

Update helloworldui-deployment-v2.yaml file with the container image you just built:

```
apiVersion: apps/v1
kind: Deployment
metadata:
...
spec:
...
  template:
    ...
    spec:
      ...
      containers:
        - image: gcr.io/MAKE_SURE_TO_REPLACE_THIS_WITH_PROJECT_ID/helloworld-ui:v2
        ...
```

Save the file, and then apply the file.

```
$ kubectl apply -f helloworldui-deployment-v2.yaml
```

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

Note: Alternatively, 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/${PROJECT_ID}/helloworld-ui:v2
```

Google Cloud Build

In the previous exercise, we built the container with regular Docker commands (`docker build ...`), and then manually pushed the image into Google Cloud Platform's Container Registry. It's also possible to defer both

steps to the server side [Cloud Build](#), which can build and push the container image without having local installation of Docker.

First, enable Container Builder API:

```
$ gcloud services enable cloudbuild.googleapis.com
```

Once the Google Cloud Build API is enabled, you can run the following command to build and push your image directly from the Cloud Build service:

```
$ cd ~/spring-boot-docker/examples/helloworld-ui
$ gcloud builds submit . -t gcr.io/${PROJECT_ID}/helloworld-ui:v3
```

Note: You can also trigger automatic container builds from a source repository event. For example, you can trigger a new container build whenever you commit code to a repository branch, or whenever you created a new release tag for a repository.

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
```

Stackdriver 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 Kubernetes Engine to run managed Kubernetes clusters, all of the logs are automatically forwarded and stored in Stackdriver Logging. You can see all the log output from the pods by navigating to **Stackdriver** → **Logging** → **Logs** in the Google Cloud console:

STACKDRIVER

Monitoring

Debug

Trace

Logging

Error Reporting

TOOLS

Logs

Logs-based Metrics

Exports

Resource Usage

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

Filter by label or text search

- ✓ Container Builder
- GCE Disk
- GCE Firewall Rule
- GCE Forwarding Rule
- GCE Health Check
- GCE Instance Group Manager
- GCE Instance Template
- GCE Project
- GCE Reserved Address
- GCE Route
- GCE Target Pool
- GCE VM Instance
- GCS Bucket
- GKE Cluster
- GKE Container
- GKE Node Pool

All logs

Any log level

Last hour

EST Container Builder GetBuild 988781e2-b717-47bb-955a-

EST 64941e367bce: Pushed

EST Container Builder GetBuild 988781e2-b717-47bb-955a-

EST Container Builder GetBuild 988781e2-b717-47bb-955a-

EST 4ca4eef4b28c: Pushed

EST Container Builder GetBuild 988781e2-b717-47bb-955a-

EST v3: digest: sha256:351 All namespace_id 80a75

All cluster_name

Search by prefix...

default

kube-system

Search by prefix...

guestbook

EST Container Builder ListBuilds builds {"@type": "type.

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

Namespaces

A single cluster can be carved into multiple namespaces. Resource names (like Deployment, Service, etc) need to be unique within a namespace, but can be reused in different namespaces. E.g., you can create a namespace *staging* and a namespace *qa*. You can deploy exactly the same application into both namespaces. Each namespace can also have its own resource constraint. E.g., *qa* namespace can be assigned 10 CPU cores limit while *staging* namespace can have more.

Create a new namespace:

```
$ kubectl create ns staging
```

See what's deployed there:

```
$ kubectl --namespace=staging get pods
No resources found.
```

Let's deploy something!

```
$ kubectl --namespace=staging apply -f \
  ~/spring-boot-docker/examples/kubernetes-1.10-simple
deployment "guestbook-service" created
service "guestbook-service" created
deployment "helloworld-service" created
service "helloworld-service" created
deployment "helloworld-ui" created
service "helloworld-ui" created
deployment "mysql" created
persistentvolumeclaim "mysql-pvc" created
service "mysql" created
deployment "redis" created
service "redis" created
```

The entire application is now deployed into staging namespace:

```
$ kubectl --namespace=staging get pods
```

guestbook-service-448345361-c0z1x	1/1	Running	0	1m
guestbook-service-448345361-tl660	1/1	Running	0	1m
helloworld-service-3269930993-4lhlt	1/1	Running	0	1m
helloworld-service-3269930993-svgpc	1/1	Running	0	1m
helloworld-ui-3002276840-43zjd	1/1	Running	0	1m
helloworld-ui-3002276840-vmnk8	1/1	Running	0	1m
mysql-4118493817-tjn78	1/1	Running	0	1m
redis-2180715406-4t6br	1/1	Running	0	1m

Find the frontend UI public IP address:

```
$ kubectl --namespace=staging get svc helloworld-ui
```

NAME	CLUSTER-IP	EXTERNAL-IP	PORT(S)	AGE
helloworld-ui	10.3.250.216	X.X.X.X	80:31116/TCP	2m

Open the browser and see the staging environment.

If you ever need to remove an entire environment under a namespace, simply delete the namespace:

```
$ kubectl delete namespace staging
namespace "staging" deleted
```

This will propagate and delete every resource under this namespace, including automatically provisioned external load balancers, and volumes. The operations are asynchronous. Even though the command line says the namespace was deleted - the resources are still being deleted asynchronously.

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:

```
$ cd ~/spring-boot-docker/examples/kubernetes-1.10
```

Edit `helloworldui-deployment-v2.yaml`, add the CPU and memory requests explicitly:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  ...
spec:
  ...
  template:
    ...
    spec:
      containers:
      - name: helloworld-ui
        image: saturnism/spring-boot-helloworld-ui:v1
        ...
        resources:
          requests:
            cpu: 200m
            memory: 128Mi
        ...
```

Save the file, and then apply the file:

```
$ kubectl apply -f helloworldui-deployment-v2.yaml
```

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 needs 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:

Edit `helloworldui-deployment-v2.yaml`, add the CPU and memory limits:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  ...
spec:
  ...
  template:
    ...
    spec:
      containers:
      - name: helloworld-ui
        image: saturnism/spring-boot-helloworld-ui:v1
        resources:
          requests:
            ...
          limits:
```



```
cpu: 500m
memory: 256Mi
...
```

Save the file, and then apply the file:

```
$ kubectl apply -f helloworldui-deployment-v2.yaml
```

Note: Wait a minute! Even though this deployment didn't have any resource limits configured, but we clearly saw that it defaulted to a request of 100m of CPU. How did that happen?

You can apply a "default" resource limits on via namespaces. We initially deployed all the workload into the default namespace. That namespace has a default limit range of 100m of CPU.

```
$ kubectl describe ns default
```

You can see the actual Limit Range configuration:

```
$ kubectl get limitrange limits -oyaml
```

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 some time 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.

Scale the Helloworld Service back down to 2 instances:

```
$ kubectl scale deployment helloworld-service --replicas=2
```

Edit `helloworldservice-deployment-v1.yaml`, add a Liveness Probe to our Helloworld Service by editing the Deployment:

```
apiVersion: apps/v1
kind: Deployment
...
```

```
spec:
...
  template:
    ...
    spec:
      ...
      containers:
      - image: saturnism/spring-boot-helloworld-service:...
        livenessProbe:
          initialDelaySeconds: 30
          httpGet:
            path: /hello/healthcheck
            port: 8080
        readinessProbe:
          httpGet:
            path: /hello/ready
            port: 8080
```

Save the file, and then apply the file:

```
$ kubectl apply -f helloworldservice-deployment-v1.yaml
```

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.

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:

```
$ cd ~/spring-boot-docker/examples/helloworld-service
```

See the `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:

Edit `helloworldservice-deployment-v1.yaml`, add the environmental variable:

```
apiVersion: apps/v1
kind: Deployment
...
spec:
...
  template:
    ...
    spec:
      ...
      containers:
        - image: saturnism/spring-boot-helloworld-service:...
          env:
            - name: GREETING
              value: Hello $name from envvar!
            ...
```

Save the file, and then apply the file:

```
$ kubectl apply -f helloworldservice-deployment-v1.yaml
```

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.

Go back to the frontend, add a new message and you should be able to see the greeting message changed.

Command Line Argument

Edit `helloworldservice-deployment-v1.yaml`, add a configuration via the command line arguments. Remove the environmental variable configurations. You know the drill, edit the Deployment, and add the following section:

```
apiVersion: apps/v1
kind: Deployment
...
spec:
...
  template:
    ...
    spec:
      ...
      containers:
      - image: saturnism/spring-boot-helloworld-service:...
        env:
        --name: GREETING
        value: Hello $name from envvar!
        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.

Save the file, and then apply the file:

```
$ kubectl apply -f helloworldservice-deployment-v1.yaml
```

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

Using ConfigMap

In this section, we'll use a 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 ~/spring-boot-docker/examples/helloworld-service
$ 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 mount the configurations as files under a specific directory, e.g.,
`/etc/config/application.properties`.

Edit `helloworldservice-deployment-v1.yaml`, add volumes and volume mounts (important - indentation matters!):

```
apiVersion: apps/v1
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
    ...
```

Save the file, and then apply the file:

```
$ kubectl apply -f helloworldservice-deployment-v1.yaml
```

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-ex11l 1/1     Running   0           1m
...
```

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

```
$ kubectl exec -ti helloworld-service-... /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
```

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.

Fun thing to try, edit the content of the ConfigMap directly in Kubernetes:

```
$ 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 Updated
kind: ConfigMap
...
```

If you go back into the same pod, and see the content of `/etc/config/application.properties`, you'll see that it's also updated:

```
$ kubectl exec -ti helloworld-service-... /bin/bash
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!

Even though ConfigMap content is automatically updated in the filesystem of the container, it's not the recommended way to update configurations. To provide new configurations, it's better to create a new ConfigMap, e.g., `greeting-config-v2`, then update the deployment to point to the new configuration.

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 Deployment 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 --from-literal=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.

Edit `guestbookservice-deployment.yaml`, add a couple of Environmental Variables:

```
apiVersion: apps/v1
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
...

```

Save the file, and then apply the file:

```
$ kubectl apply -f guestbookservice-deployment.yaml
```

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

Note: Keep in mind that Secrets are not encrypted at rest by default. It's good to keep in mind that there are other credential alternatives. For example, you can [integrate with Vault](#), or use [Sealed Secrets](#).

Autoscaling

Duration: 5:00

Kubernetes has built-in Horizontal Pod Autoscaling based on CPU utilization (and custom metrics!). We will cover autoscaling based on CPU utilization in this lab.

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](#).

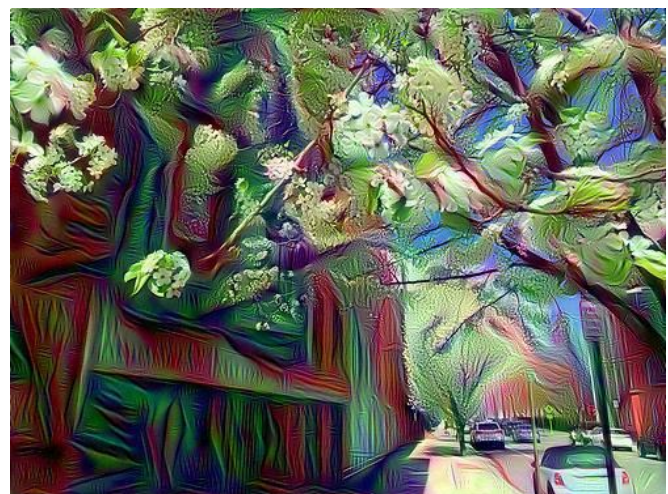
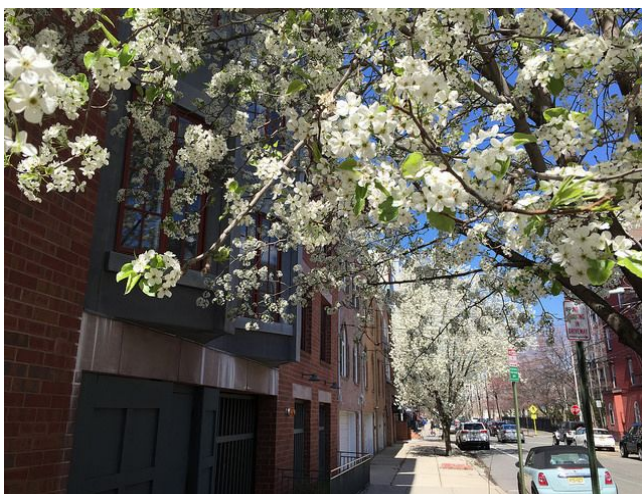
Managing Batched / Run-Once Jobs

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 Replica Sets and Deployments to run a job that is expected to 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 [DeepDream](#) to produce a dreamy picture. The job will retrieve an image from the web, processes it with 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:

```
$ export PROJECT_ID=$(gcloud config get-value core/project)
$ 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
```

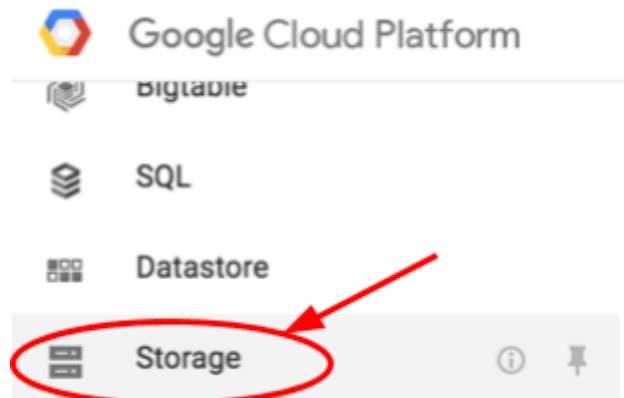
Note: You can also create a Job descriptor as well. Read the [Kubernetes Jobs Guide](#) to learn how you can write a YAML file to schedule a job.

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

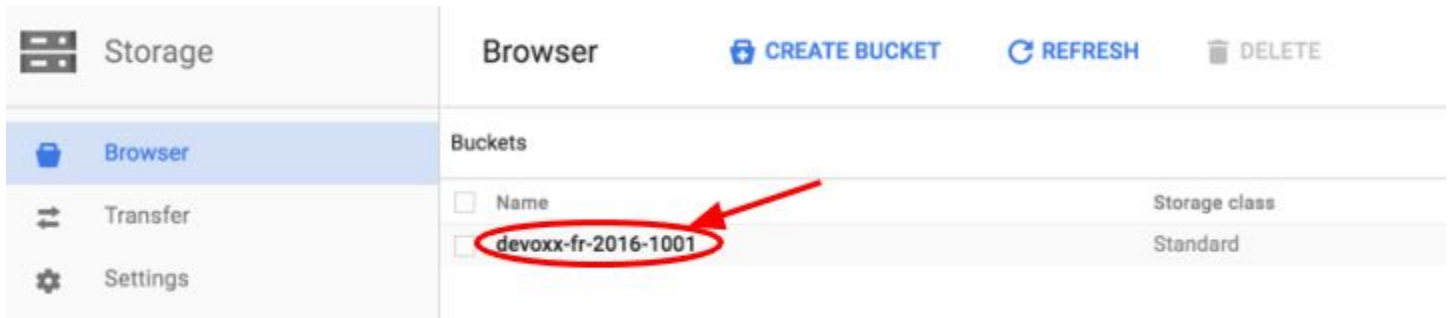
```
$ kubectl get jobs
NAME            DESIRED    SUCCESSFUL    AGE
deepdream-1     1          0             13m
```

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

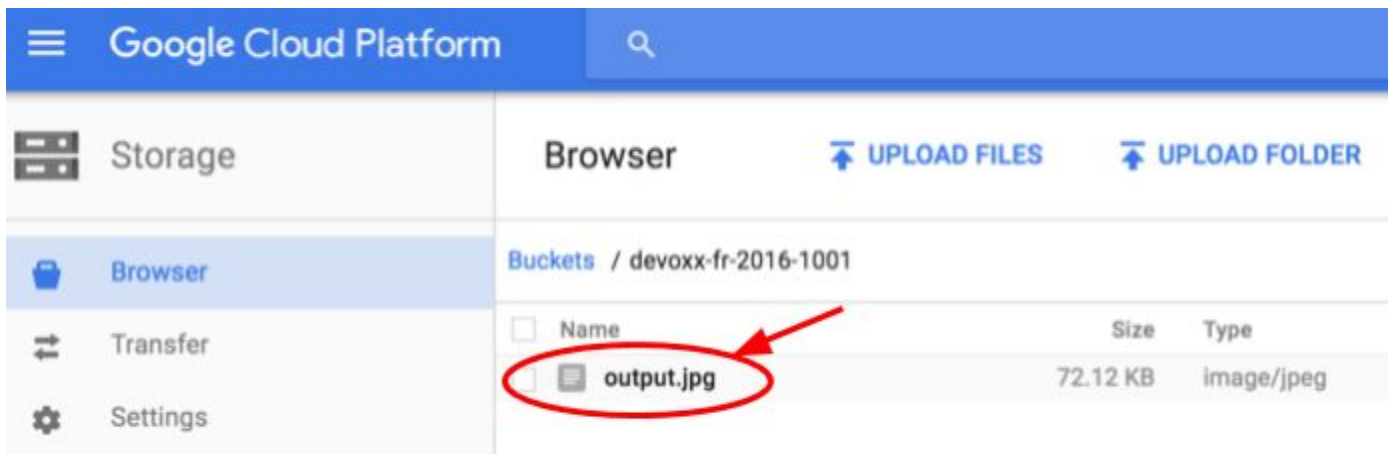
First, navigate to **Storage > Browser**:



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.

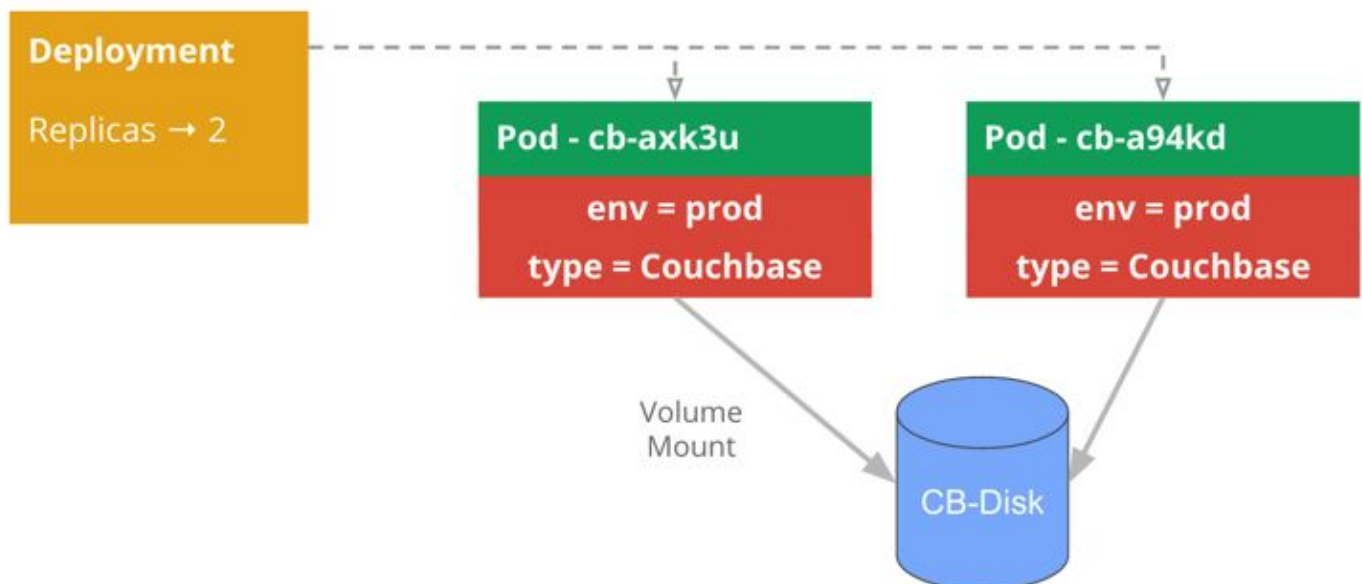
Note: Kubernetes has support for Cron (scheduled) Jobs. Think of it as a distributed crontab. You can learn more in the [Kubernetes Cron Job Guide](#).

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.

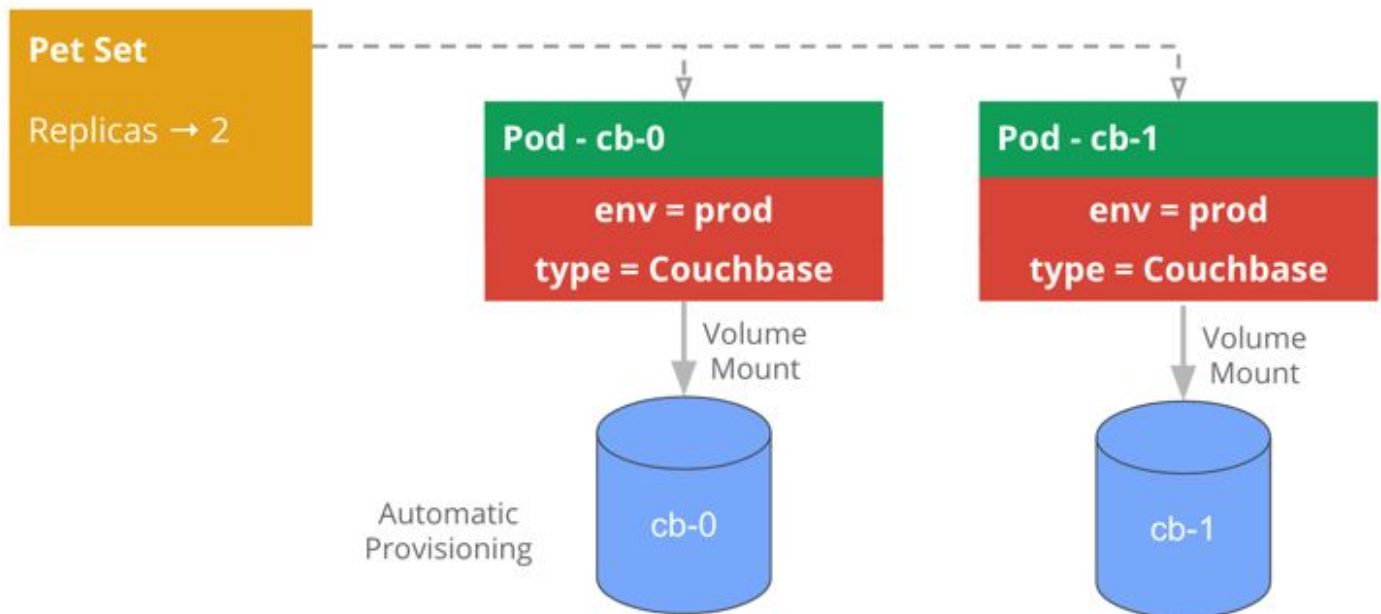
If each of your application instance is stateful (i.e., stores instance-specific data), you'll run into issues quickly if you use a normal Kubernetes Deployment, because each of the replicas will be pointing to exactly the same persistent volume for storage - and it may not work.

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



Not only that, each Pod will have an 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:



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-statefulset/couchbase-service.yml
```

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

- The first service has a ClusterIP attribute 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-statefulset/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 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.

Validate that all the pods were started successfully. Once started, 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 scale the StatefulSet 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 delete all of the Pod instances as well as all the storage simply by deleting the StatefulSet:

```
$ kubectl delete statefulset couchbase
```

Lastly, delete the services too!

```
$ kubectl delete svc couchbase couchbase-ui
```

Note: If you are interested in deploying Akka in Kubernetes, you can read [Clustering Akka in Kubernetes with Statefulset and Deployment](#) blog - it utilizes both Deployment and StatefulSet.

Deploy Whatever You Want

Deploy a Monolith - Pet Clinic

The best way validate what you've learned is to put them into practice, from scratch! Here is a challenge - configure and deploy [Spring Pet Clinic](#) in Kubernetes.

There is a good Spring Pet Clinic Docker container on Docker Hub: [anthonydahanne/spring-petclinic](#), I recommend start from there.

You should create the necessary resource YAML files (hint, look at files in `~/spring-boot-docker/examples/kubernetes-1.10`):

For MySQL:

- `mysql-deployment.yaml`
- `mysql-pvc.yaml`
- `mysql-service.yaml`

For the Application:

- `petclinic-deployment.yaml`
- `petclinic-service.yaml`

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.
- [Custom resource definitions](#) - You can create your own resource descriptors and provision your own controllers!
- [Helm](#) - A package manager for Kubernetes. There are many pre-packaged deployments you can easily provision into a Kubernetes cluster.
- Security features: [Role-Based Access Control](#), [Network Policy](#) and [Pod Security Policy](#)

Cleanup

Duration: 5:00

If you are using your own account, 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.

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.

Your account has a default quota of 24 CPU cores, and 8 IP addresses by default. To do the next portion of the lab, you'll need to remove the existing resources first in order to ensure you don't exceed the quota.

First, let's undeploy everything from the guestbook cluster:

```
$ cd ~/spring-boot-docker/examples/kubernetes-1.10
$ kubectl delete -f .
```

Next, delete the cluster altogether:

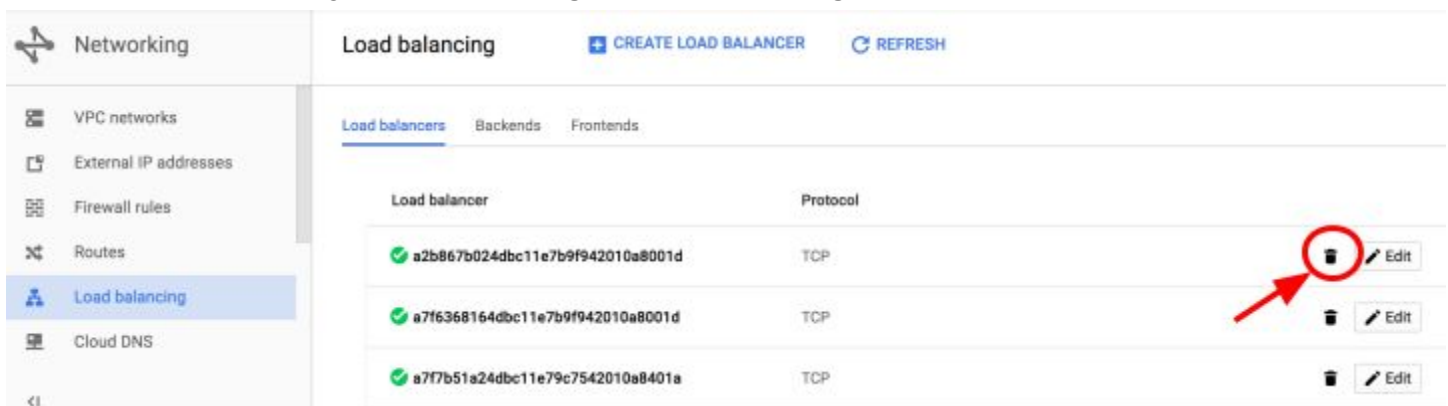
```
$ gcloud container clusters delete guestbook
The following clusters will be deleted.
- [guestbook] in [us-central1-c]
Do you want to continue (Y/n)? Y
Deleting cluster guestbook...done.
Deleted [https://container.googleapis.com/v1/projects/.../guestbook].

$ gcloud compute disks delete mysql-disk

$ 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>/...
```

Just to be sure, also navigate to **Networking** → **Load Balancing** and *delete every load balancer*.



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> 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

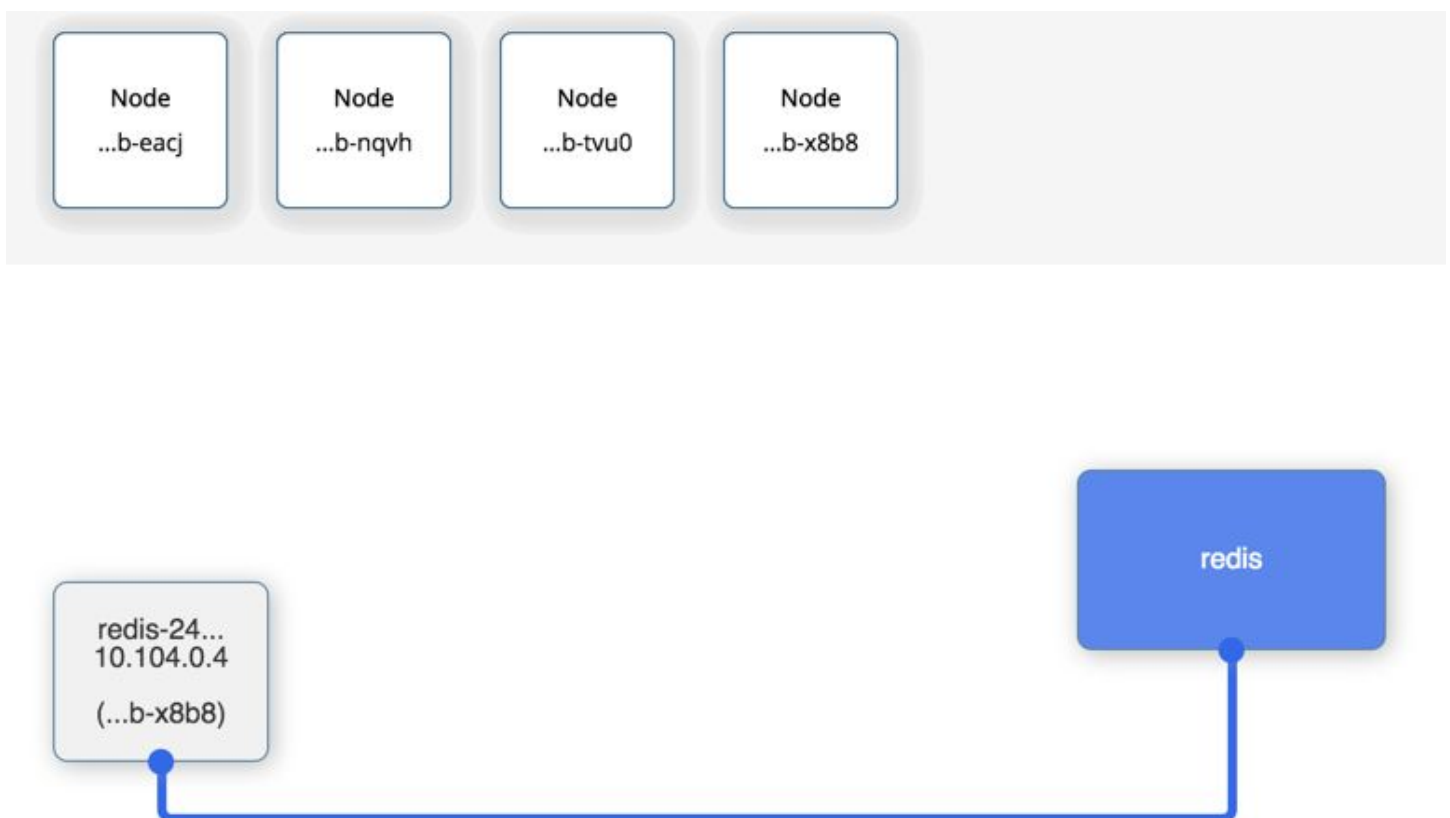
Download the open source version, build it and deploy a cluster yourself with the kubernetes tools.

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> 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 :)

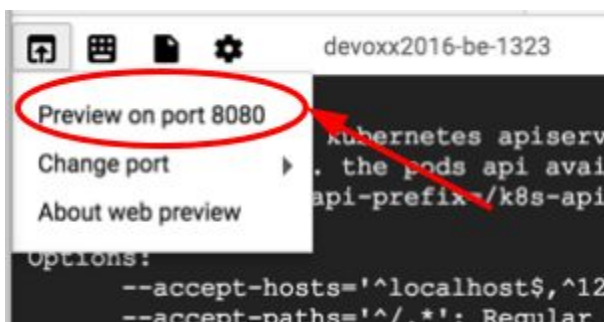
You can install the visualizer for this lab if you like - but it'll take a little time and extra steps. First, clone the GitHub repository, and let's go into it:

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

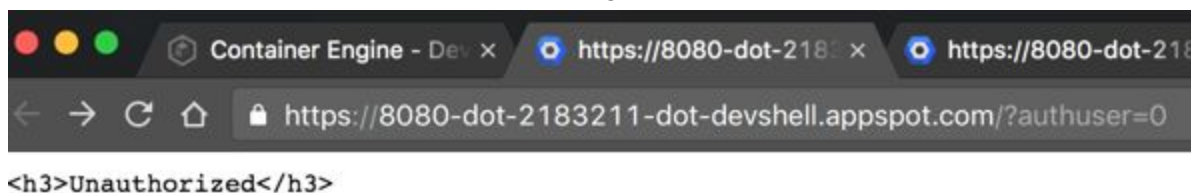
You can then start a local Kubernetes proxy that will forward requests from the proxy to the Kubernetes server via a secure proxy. It's important to know that you can issue any Kubernetes API calls to the proxy without additional authentication:

```
$ 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:



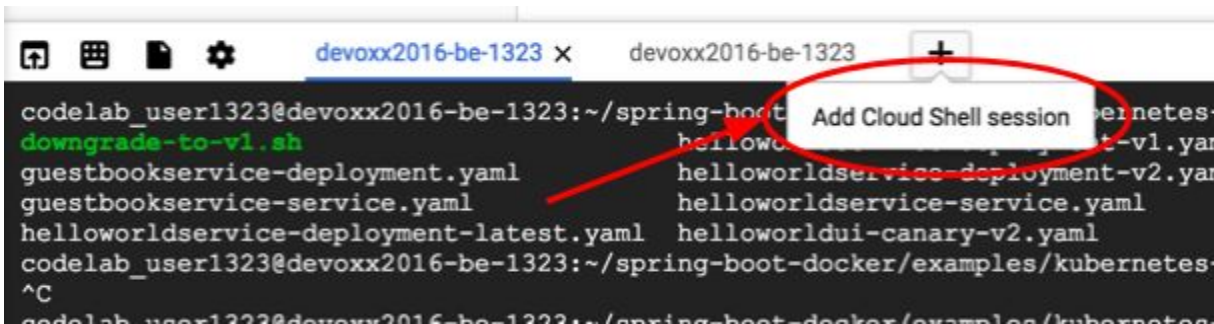
You will first see an Unauthorized error message:



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](https://groups.google.com/forum/#!forum/google-containers)
- twitter: [@kubernetesio](https://twitter.com/kubernetesio)
- IRC: #google-containers on freenode

Minikube

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

Google Kubernetes Engine

- <https://cloud.google.com/kubernetes-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 Kubernetes Engine Lab](#)
- [RedHat Developer Kubernetes Lab](#)

Run a Container

Duration: 10:00

We'll start off with something straight forward. What if you already have a container built and ready to be used? You can deploy a container image into a cluster of machines, and scale them, easily with Kubernetes.

For example, to run a container image you built:

```
$ kubectl run myapp --image=YOUR_DOCKER_IMAGE
// Example with NGINX:
// kubectl run myapp --image=nginx
```

You can scale the number of instances of your application:

```
$ kubectl scale deployment myapp --replicas=2
```

Check the status of all of the instances and make sure all instances are in the RUNNING status:

```
$ kubectl get pods
```

To make this application accessible from the public Internet, you can provision a load balancer:

```
$ kubectl expose deployment myapp \
  --type=LoadBalancer --port=80 --target-port=YOUR_APP_PORT
// Example with NGINX:
// kubectl expose deployment myapp --type=LoadBalancer --port=80 --target-port=80
```

You can retrieve the public IP address by checking the External IP of the service. It will take a few minutes to provision the public IP. If you see the External IP as Pending, check again in a minute or two:

```
$ kubectl get svc myapp
```

NAME	CLUSTER-IP	EXTERNAL-IP	PORT(S)	AGE
myapp	10.15.253.160	XXX.XXX.XXX.XXX	80:30735/TCP	29s

Use the browser to browse to the newly provisioned IP address and you should see the application running.

Finally, you can un-deploy both the app and the load balancer:

```
$ kubectl delete deployment/myapp svc/myapp
```

What just happened?! Let's go through this in detail in the rest of the lab with deploying something more complex.