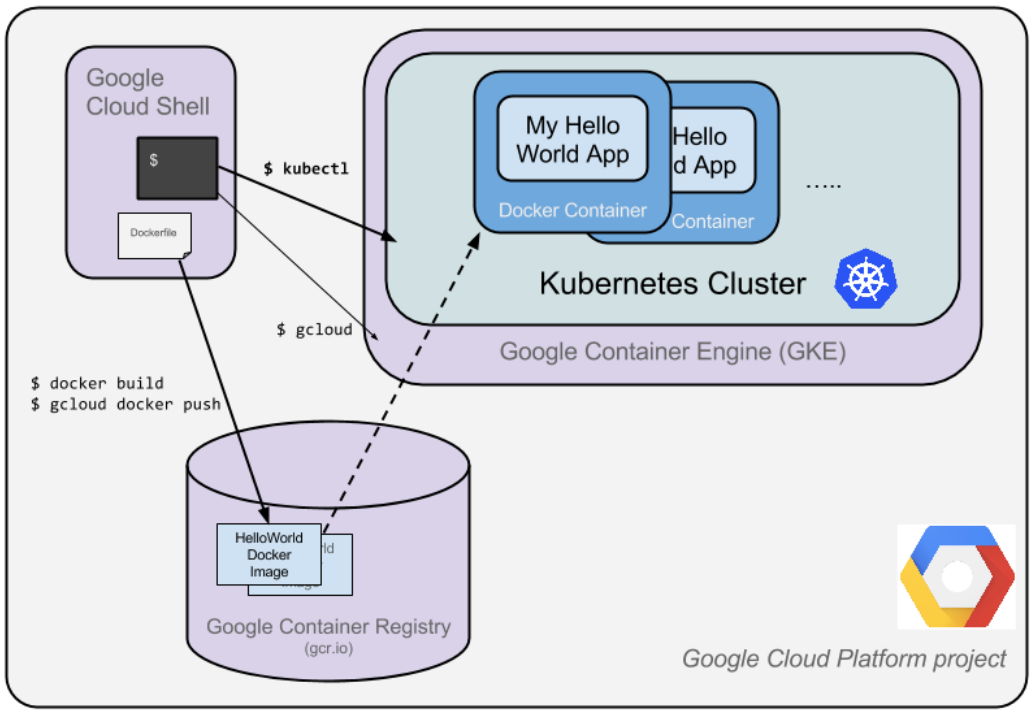
https://codelabs.developers.google.com/codelabs/cloud-springboot-kubernetes/index.html?index=..%2F..%2Fspringone#0

1. Overview

Kubernetes is an open source project which can run in many different environments, from laptops to high-availability multi-node clusters, from public clouds to on-premise deployments, from virtual machines to bare metal.
In this lab, you deploy a simple Java web-based application (using Spring Boot) to Kubernetes running on Kubernetes Engine.
The goal of this codelab is for you to run your web application with as a replicated application running on Kubernetes. You take code that you have developed on your machine, turn it into a Docker container image, and then run that image on Kubernetes Engine.
Here's a diagram of the various parts in play in this codelab to help you understand how pieces fit together. Use this as a reference as you progress through the codelab; it should all make sense by the time you get to the end (but feel free to ignore this for now).



For the purpose of this codelab, using a managed environment such as Kubernetes Engine (a Google-hosted version of Kubernetes running on Compute Engine) allows you to focus more on experiencing Kubernetes rather than setting up the underlying infrastructure.
If you are interested in running Kubernetes on your local machine, such as a development laptop, you should probably look into Minikube. This offers a simple setup of a single node kubernetes cluster for development and testing purposes. You can use Minikube to go through this codelab if you wish.
This tutorial uses the sample code from the Spring Boot Getting Started guide.

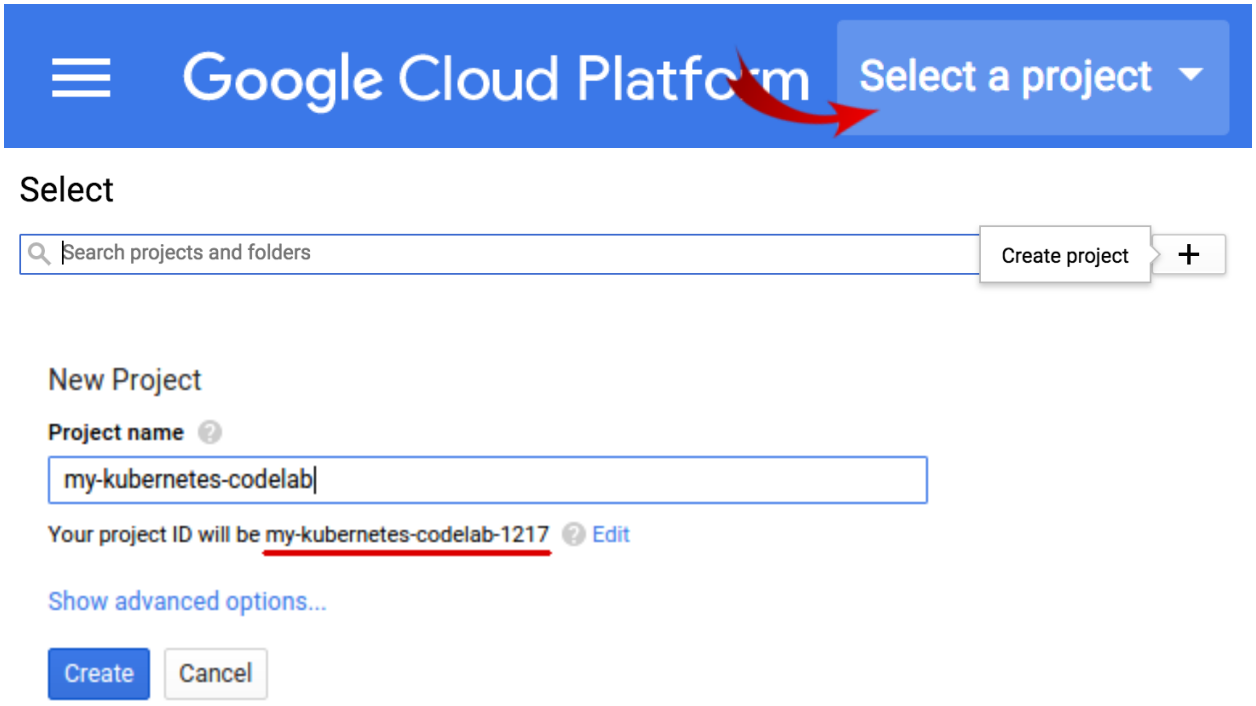
What you'll learn

- How to package a simple Java application as a Docker container.
- How to create your Kubernetes cluster on Kubernetes Engine.
- How to deploy your Java application into Kubernetes on Kubernetes Engine
- How to scale up your service and roll out an upgrade.
- How to access Kubernetes Graphical dashboard.

2. Setup and Requirements

Self-paced environment setup

If you don't already have a Google Account (Gmail or Google Apps), you must create one. Sign-in to Google Cloud Platform console (console.cloud.google.com) and create a new project:



Remember the project ID, a unique name across all Google Cloud projects (the name above has already been taken and will not work for you, sorry!). It will be referred to later in this codelab as PROJECT_ID.

Next, you'll need to enable billing in the Developers Console in order to use Google Cloud resources. Running through this codelab shouldn't cost you more than a few dollars, but it could be more if you decide to use more resources or if you leave them running (see "cleanup" section at the end of this document). Google Kubernetes Engine pricing is documented here.
New users of Google Cloud Platform are eligible for a $300 free trial.

3. Use OpenJDK 8

Google Cloud Shell has both Java 7 and Java 8 installed. It uses Java 7 by default. Let's switch to use Java 8 instead. In the Cloud Shell, use update-alternative command to change the default Java version (make sure you select the java-8-openjdk option by typing "2"):

```
$ sudo update-alternatives --config javac
There are 2 choices for the alternative javac (providing /usr/bin/javac).
  Selection    Path                                            Priority   Status
------------------------------------------------------------
* 0            /usr/lib/jvm/java-7-openjdk-amd64/bin/javac   ...
  1            /usr/lib/jvm/java-7-openjdk-amd64/bin/javac   ...
  2            /usr/lib/jvm/java-8-openjdk-amd64/bin/javac   ...
Press enter to keep the current choice[*], or type selection number: 2
update-alternatives: using /usr/lib/jvm/java-8-openjdk-amd64/bin/javac to provide /usr/bin/javac
(javac) in manual mode

$ sudo update-alternatives --config java
There are 2 choices for the alternative java (providing /usr/bin/java).
  Selection    Path                                            Priority   Status
------------------------------------------------------------
* 0            /usr/lib/jvm/java-7-openjdk-amd64/jre/bin/java   ...
  1            /usr/lib/jvm/java-7-openjdk-amd64/jre/bin/java   ...
  2            /usr/lib/jvm/java-8-openjdk-amd64/jre/bin/java   ...
Press enter to keep the current choice[*], or type selection number: 2
update-alternatives: using /usr/lib/jvm/java-8-openjdk-amd64/jre/bin/java to provide /usr/bin/java
(java) in manual mode
```

4. Get the Spring Boot Getting Started Example source code

After Cloud Shell launches, you can use the command line to clone the example source code in the home directory:

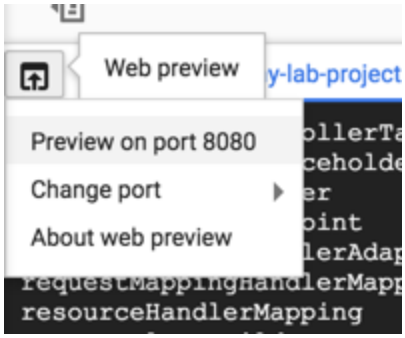
```
$ git clone https://github.com/spring-guides/gs-spring-boot.git
$ cd gs-spring-boot/complete
```

5. Run the Application Locally

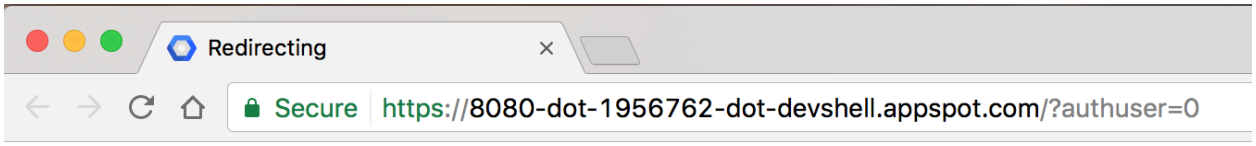
You can start the Spring Boot application normally with the Spring Boot plugin:

```
$ ./mvnw -DskipTests spring-boot:run
```

Once the application started, click on the Web Preview icon in the Cloud Shell toolbar and choose preview on port 8080.



A tab in your browser opens and connects to the server you just started.




```
Hello!
```

6. Package the Java application as a Docker container

Next, prepare your app to run on Kubernetes. The first step is to define the container and its contents.
First, create the JAR deployable for the application

```
$ ./mvnw -DskipTests package
```

Then, create a Dockerfile:

```
$ touch Dockerfile
```

Add the following to Dockerfile using your favorite editor (vim, nano,emacs or Cloud Shell's code editor):

```
FROM openjdk:8
COPY target/gs-spring-boot-0.1.0.jar /app.jar
EXPOSE 8080/tcp
ENTRYPOINT ["java", "-jar", "/app.jar"]
```

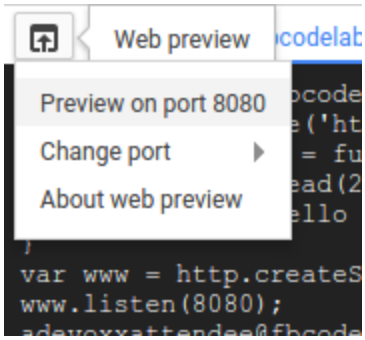
The Dockerfile shown above builds on the OpenJDK image, which is already configured to have OpenJDK pre-installed and can run your JAR.
Save this Dockerfile and build this image by running this command (make sure to replace PROJECT_ID with yours) :

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

Once this completes (it'll take some time to download and extract everything) you can test the image locally with the following command which will run a Docker container as a daemon on port 8080 from your newly-created container image:

```
$ docker run -ti --rm -p 8080:8080 gcr.io/PROJECT_ID/hello-java:v1
```

And again take advantage of the Web preview feature of CloudShell :



You should see the default page in a new tab. Once you verify that the app is running fine locally in a Docker container, you can stop the running container by pressing Ctrl+C.
Now that the image works as intended you can push it to the Google Container Registry, a private repository for your Docker images accessible from every Google Cloud project (but also from outside Google Cloud Platform) :

```
$ gcloud docker -- push gcr.io/PROJECT_ID/hello-java:v1
```

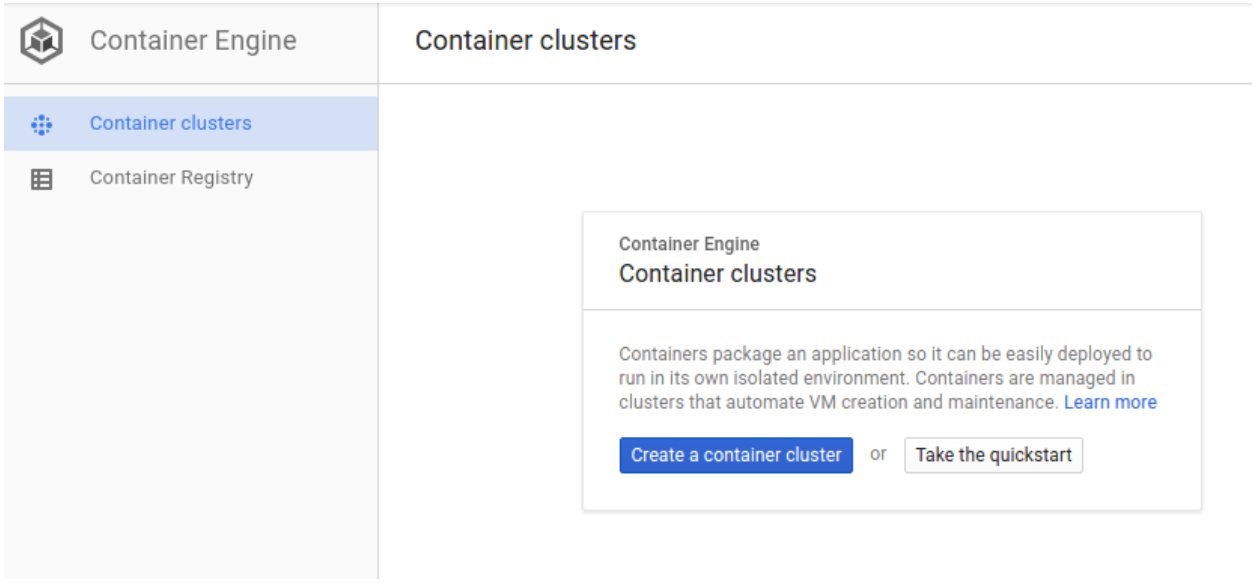
If all goes well and after a little while you should be able to see the container image listed in the console: *Tools > Container Registry*. At this point you now have a project-wide Docker image available which Kubernetes can access and orchestrate as you'll see in a few minutes.

Note that while here we used a generic domain for the registry (gcr.io), you can also be more specific about which zone and bucket to use. Details are documented here: https://cloud.google.com/container-registry/docs/#pushing_to_the_registry

If you're curious, you can navigate through the container images as they are stored in Google Cloud Storage by following this link: https://console.cloud.google.com/storage/browser/ (the full resulting link should be of this form: https://console.cloud.google.com/project/*PROJECT_ID*/storage/browser/).

7. Create your cluster

Ok, you are now ready to create your Kubernetes Engine cluster but before that, navigate to the Google Kubernetes Engine section of the web console and wait for the system to initialize (it should only take a few seconds).



A cluster consists of a Kubernetes master API server managed by Google and a set of worker nodes. The worker nodes are Compute Engine virtual machines. Let's use the gcloud CLI from your CloudShell session to create a cluster with two n1-standard-1 nodes (this will take a few minutes to complete):

```
$ gcloud container clusters create hello-java-cluster \
  --num-nodes 2 \
  --machine-type n1-standard-1 \
  --zone us-central1-c
```

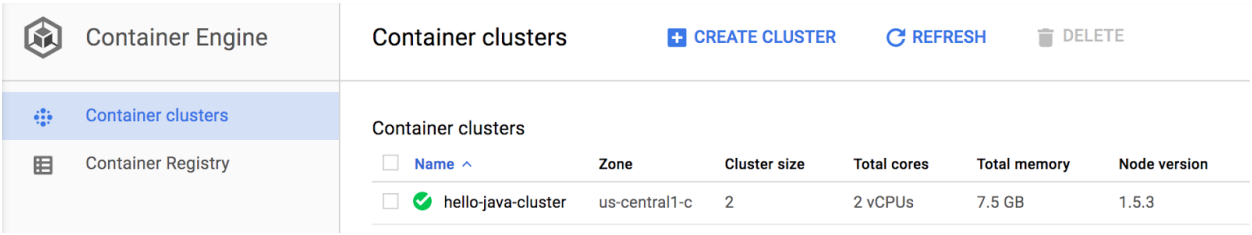
In the end, you should see the cluster created.

```
Creating cluster hello-java-cluster...done.
Created [https://container.googleapis.com/v1/projects/...].
kubeconfig entry generated for hello-dotnet-cluster.
NAME                ZONE           MASTER_VERSION
hello-java-cluster  us-central1-c  ...
```

Alternatively, you could create this cluster via the Console shown above: *Compute > Kubernetes Engine > Container Clusters > Create a container cluster*.

You can create the cluster in another Zone but it is recommended to keep it on the same region as the storage bucket used by the container registry (see previous step).

You should now have a fully-functioning Kubernetes cluster powered by Google Kubernetes Engine:



It's now time to deploy your own containerized application to the Kubernetes cluster! From now on you'll use the kubectl command line (already set up in your Cloud Shell environment). The rest of this codelab requires both the Kubernetes client and server version to be 1.2 or above. kubectl version will show you the current version of the command.

8. Deploy your application to Kubernetes

A Kubernetes deployment can create, manage, and scale multiple instances of your application using the container image you've just created. Let's deploy one instance of your application into Kubernetes using the kubectl run command (replace PROJECT_ID with your own project name):

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

To view the deployment you just created, simply run:

```
$ kubectl get deployments
NAME         DESIRED   CURRENT   UP-TO-DATE   AVAILABLE   AGE
hello-java   1         1         1            1           37s
```

To view the application instances created by the deployment, run this command:

```
$ kubectl get pods
NAME                          READY     STATUS    RESTARTS   AGE
hello-java-714049816-ztzrb    1/1       Running   0          57s
```

At this point you should have your container running under the control of Kubernetes but you still have to make it accessible to the outside world.

9. Allow external traffic

By default, the pod is only accessible by its internal IP within the cluster. In order to make the hello-java container accessible from outside the kubernetes virtual network, you have to expose the pod as a kubernetes service.

From Cloud Shell you can expose the pod to the public internet with the kubectl expose command combined with the --type=LoadBalancer flag. This flag is required for the creation of an externally accessible IP :

```
$ kubectl expose deployment hello-java --type=LoadBalancer
```

The flag used in this command specifies that you'll be using the load-balancer provided by the underlying infrastructure (in this case the Compute Engine Load Balancer). Note that you expose the deployment, and not the pod directly. This will cause the resulting service to load balance traffic across all pods managed by the deployment (in this case only 1 pod, but you will add more replicas later).
The Kubernetes master creates the load balancer and related Compute Engine forwarding rules, target pools, and firewall rules to make the service fully accessible from outside of Google Cloud Platform.
To find the publicly-accessible IP address of the service, simply request kubectl to list all the cluster services:

```
$ kubectl get services
NAME         CLUSTER-IP     EXTERNAL-IP      PORT(S)    AGE
Hello-java   10.3.253.62    aaa.bbb.ccc.ddd  8080/TCP   1m
kubernetes   10.3.240.1     <none>           443/TCP    5m
```

The EXTERNAL-IP may take several minutes to become available and visible. If the EXTERNAL-IP is missing, wait a few minutes and try again.

Note there are 2 IP addresses listed for your service, both serving port 8080. One is the internal IP that is only visible inside your cloud virtual network; the other is the external load-balanced IP. In this example, the external IP address is aaa.bbb.ccc.ddd.
You should now be able to reach the service by pointing your browser to this address: http://<EXTERNAL_IP>:8080

10. Scale up your service

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

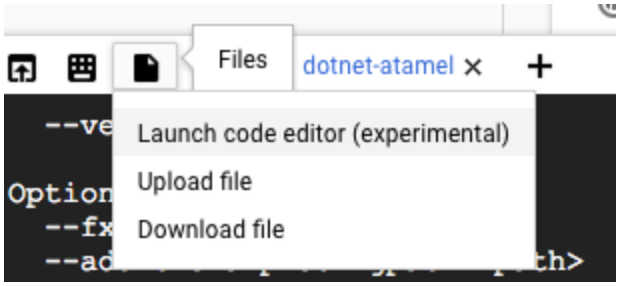
```
$ kubectl scale deployment hello-java --replicas=3
deployment "hello-java" scaled
```

```
$ kubectl get deployment
NAME         DESIRED   CURRENT   UP-TO-DATE   AVAILABLE   AGE
hello-java   3         3         3            3           22m
```

Note the declarative approach here - rather than starting or stopping new instances you declare how many instances should be running at all time. Kubernetes reconciliation loops simply make sure the reality matches what you requested and takes action if needed.

11. Roll out an upgrade to your service

At some point the application that you've deployed to production will require bug fixes or additional features. Kubernetes is here to help you deploy a new version to production without impacting your users.
First, let's modify the application. Open the code editor from Cloud Shell.



Navigate to /gs-spring-boot/complete/src/main/java/hello/HelloController.java
, and update the value of the response:

```
package hello;

import org.springframework.web.bind.annotation.RestController;
import org.springframework.web.bind.annotation.RequestMapping;

@RestController
public class HelloController {
    @RequestMapping("/")
    public String index() {
        return "Greetings from Google Kubernetes Engine!";
    }
}
```

Then rebuild the application with the latest changes:

```
$ ./mvnw -DskipTests package
```

Then build a new version of the container image:

```
$ docker build -t gcr.io/PROJECT_ID/hello-java:v2 .
```

And push the image into the container image registry:

```
$ gcloud docker -- push gcr.io/PROJECT_ID/hello-java:v2
```

Building and pushing this updated image should be much quicker as we take full advantage of caching.

You're now ready for Kubernetes to smoothly update your replication controller to the new version of the application. In order to change the image label for your running container, you need to edit the existing hello-java deploymentand change the image from gcr.io/PROJECT_ID/hello-java:v1 to gcr.io/PROJECT_ID/hello-java:v2.
You can use kubectl set image command to ask Kubernetes to deploy the new version of your application across the entire cluster one instance at a time with rolling update:

```
$ kubectl set image deployment/hello-java \
  hello-java=gcr.io/PROJECT_ID/hello-java:v2
```

```
deployment "hello-java" image updated
```

While this is happening, the users of the services might see interruption. But that is because there was no health check configured. That's a slightly more advanced configuration - we won't add health check in this lab.

Check http://EXTERNAL_IP:8080 again to see that it's returning the new response.

12. Roll back

Oops - did you make a mistake with a new version of the application? Perhaps the new version contained an error and you need to rollback quickly. With Kubernetes, you can roll back to the previous state easily. Let's rollback the application by running:

```
$ kubectl rollout undo deployment/hello-java
```