**Continuous integration and deployment of a Spring microservice application to Google Cloud Kubernettes Engine**

In this article we’ll build a continuous integration and deployment pipeline into Kubernetes running on Google Cloud platform. We’ll use (a) gitlab.com for the pipeline, (b) Google Cloud for the Kubernetes, and (c) Spring for the microservices application. This article assumes that you have some familiarity with Google Cloud Platform, Kubernetes, Linux and Spring. Let’s get started!

**Core Setup**

1. Download the code from github at this link.
2. Create a new project on gitlab.com and connect your local code to it, as such.

cd folder\_containing\_downloaded\_code

git init

git remote add origin https://gitlab.com/<me>/<repo>.git

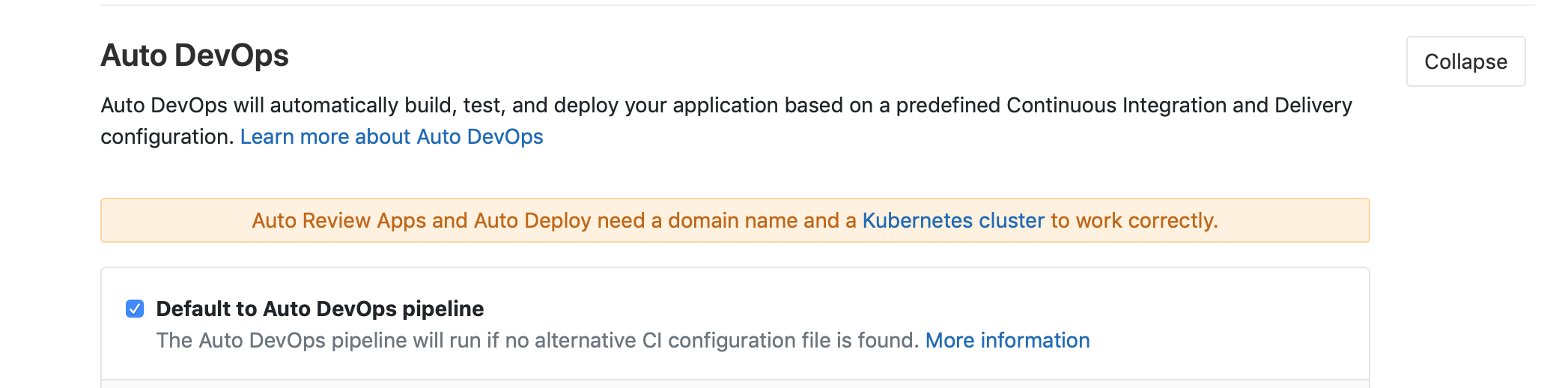
git add .

git commit -m "Initial commit"

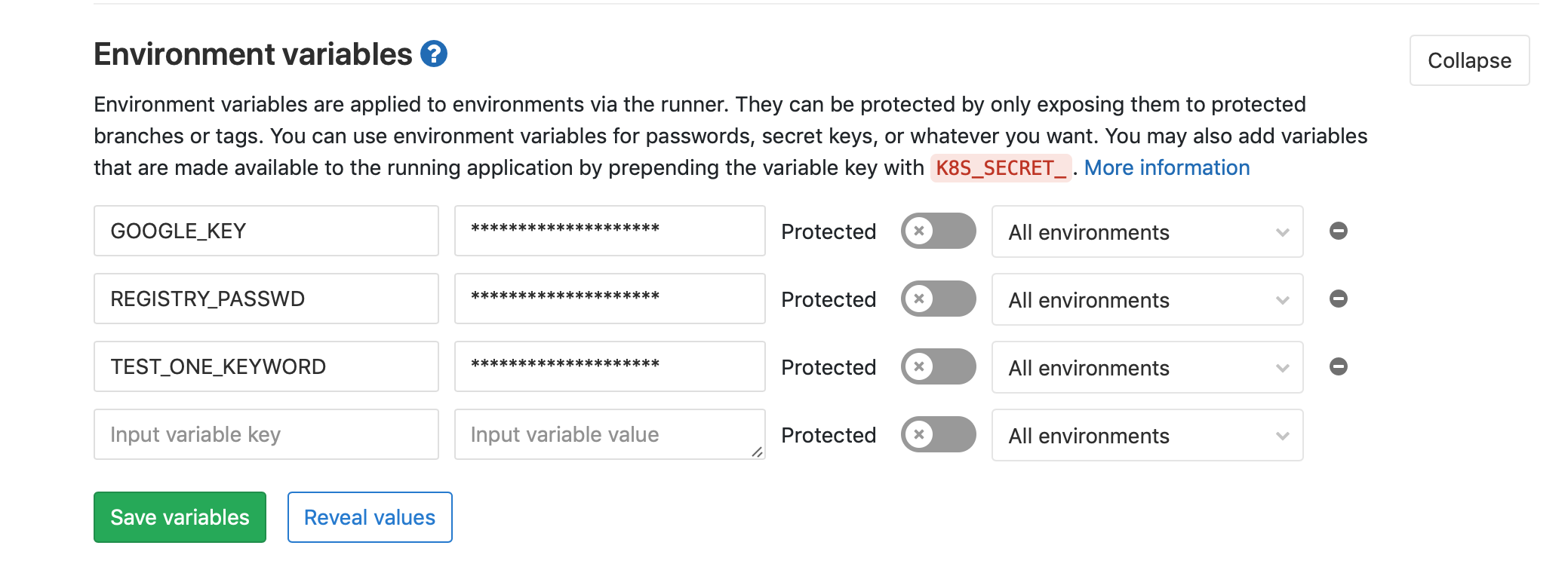
git push -u origin master

where <me> and <repo> are replaced with the correct tags for your repository.

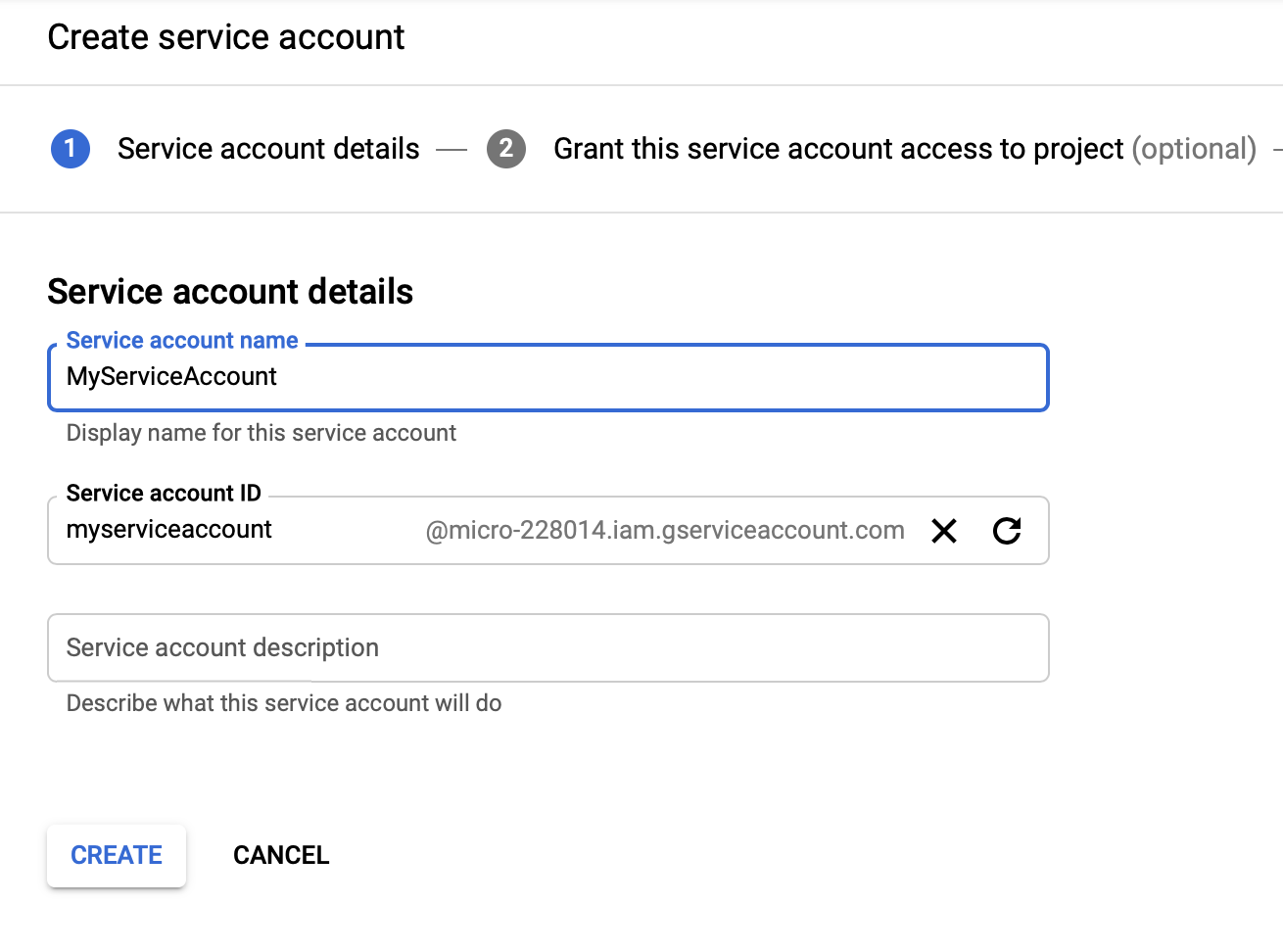
1. Turn on Auto-Devops. This will cause Gitlab to run our pipeline on each checkin to the repository. In your Gitlab project, go to Settings 🡪 CI/CD and expand out the Auto DevOps section. Check the box as shown.



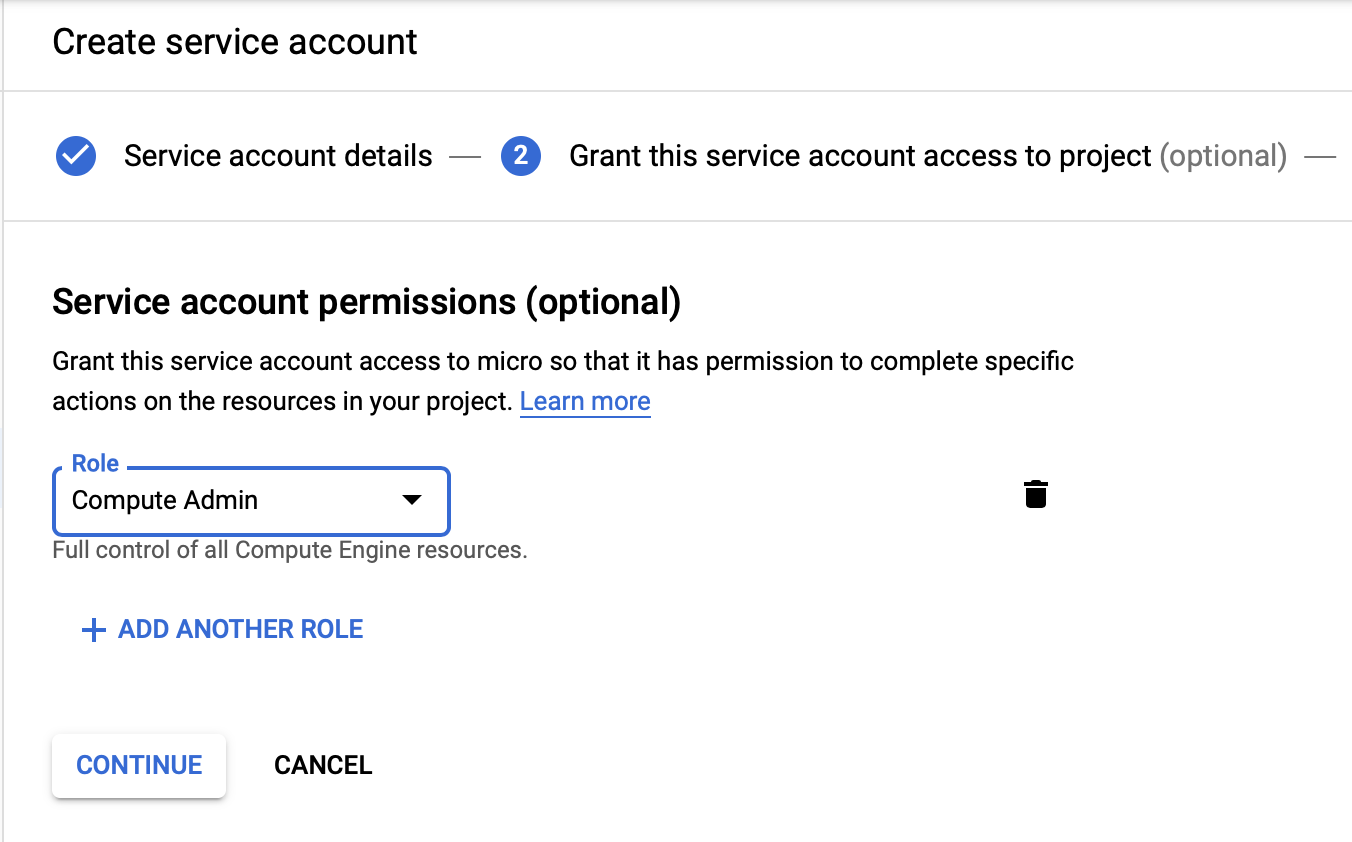
1. Setup Gitlab CI/CD environment variables. We’ll need 3 variables to be stored here, (a) a Google Key to enable the pipeline in Gitlab to execute commands against our Kubernetes cluster on our behalf, (b) a password to the Gitlab repository where we store the containers built by our pipeline, and (c) a keyword for use in the smoke test portion of our pipeline. Create a variable called **REGISTRY\_PASSWD** and enter your Gitlab account password here. Create another variable called **TEST\_ONE\_KEYWORD** and enter the value “alive” in the box to the right. Also create a variable called **GOOGLE\_KEY**, but don’t enter a value just yet. We need some information from Google Cloud Platform to complete that item, and will do that next.



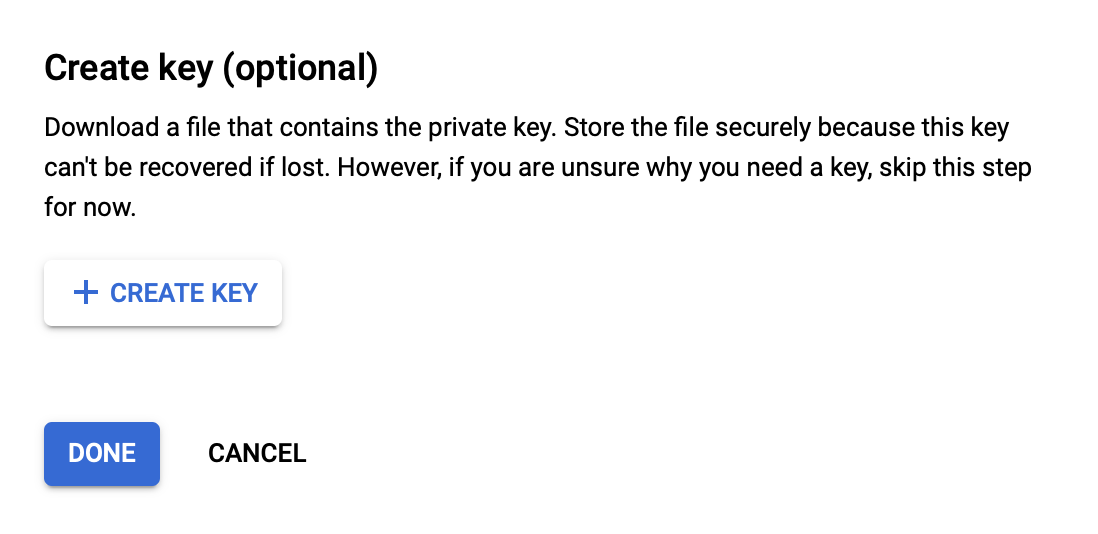
1. **The Google Key**. This is the key representing a service account in Google Cloud Platform which is authorized to do specific actions relative to the Kubernetes cluster. It represents a ***Service Account*** in GCP. For this step, you’ll need a Google Cloud Platform account and project (if you don’t already have one, follow [these](https://developers.google.com/maps/premium/devconsole-access#creating_a_google_account) steps to get that in place). First, go to your cloud console for the project you want to use, and select IAM \* Admin 🡪 Service Account. Click Create Service Account.



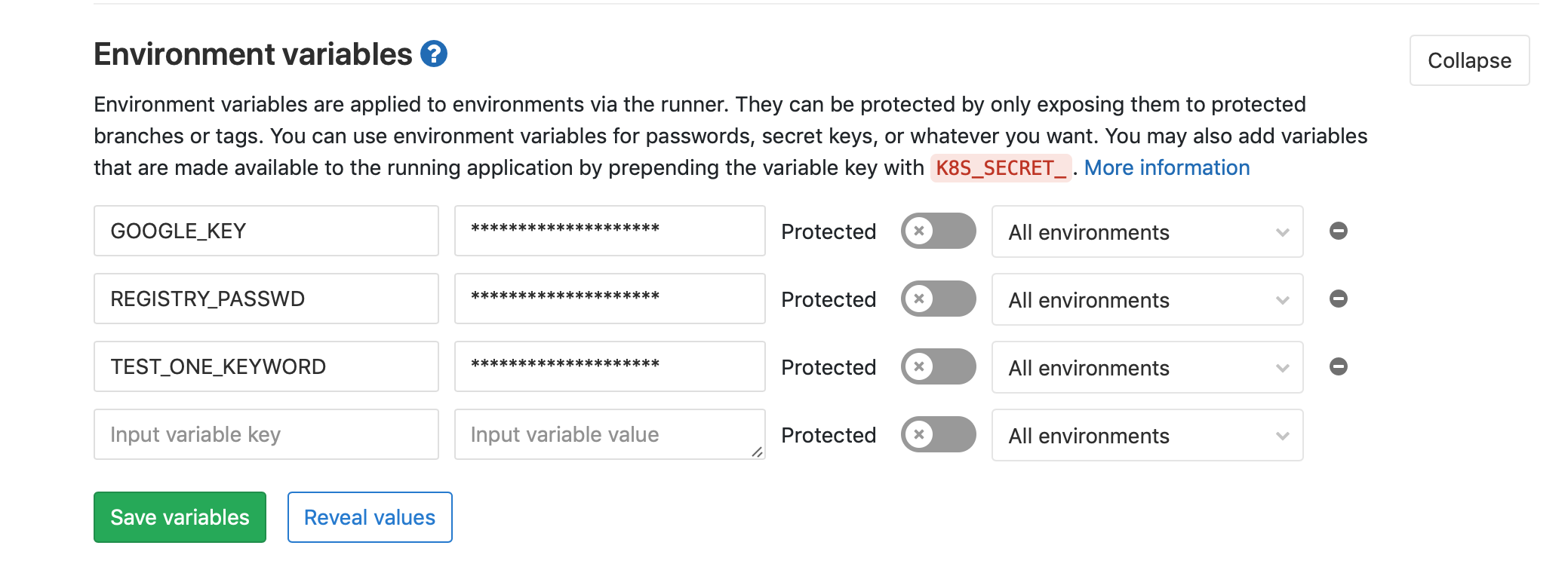
Give this account some permissions. We will give it Compute Engine admin, so it can update our cluster as part of the pipeline.



Click Continue and then click Create Key. Select JSON in the next screen, and save the file locally.



That’s it on that. Now we can add it to our Gitlab environment variables. Go back to Gitlab 🡪 Settings 🡪 CICD 🡪 Environment Variables and set the value for the variable GOOGLE\_KEY by pasting in the entire contents of the JSON file into the box next to it. Click Save.



**The Pipeline**

We now have credentials in place for operating a pipeline of code delivery into Google Cloud Kubernetes Engine. Let’s build the actual pipeline now. A Gitlab pipeline is specified by a .gitlab-ci.yml file which must be located in the root of the repository. If Gitlab finds that file, it will kick off the pipeline on each commit to the repository.

Here is the front of that file. Notice that Gitlab pipelines use docker containers to run each stage, so we need Docker in Docker as our primary service.



The pipeline has 6 stages, all executed in sequence if the preceding stage was successful. Each one runs in a docker container, using an image we specify for the type of work in that stage.

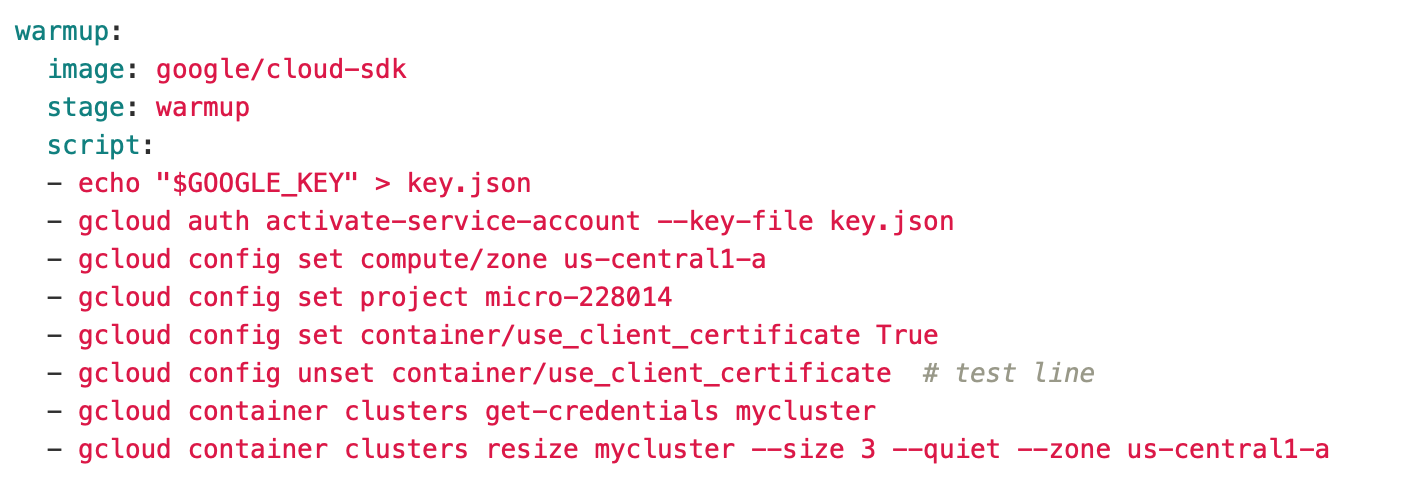
The first 2 stages build, package (containerize) and then store the application on Gitlab’s repository.



Once our code is built and successfully packaged as a Docker container and stored in our repository, we are ready to prepare our test environment. Notice that our pipeline exists entirely to test a code checkin against an integration environment, not just a simple set of jUnit tests. To do this, we spin up a cluster and deploy/test our new code there.

So, we have a warmup stage. Here, we are scaling up our deployment and testing cluster. Our cluster is always alive, but is kept at 0 nodes when not in use, to save resource cost. In GCP Kubernetes Engine the nodes are actually VMs, and while they are running we incur cost. So, when we are not testing new code, we reduce the nodes to 0, so no VMs are running. The cluster is still alive, it just have far fewer resources allocated.

This stage will use a Google Cloud console docker image, since we will be executing Google Cloud commands.



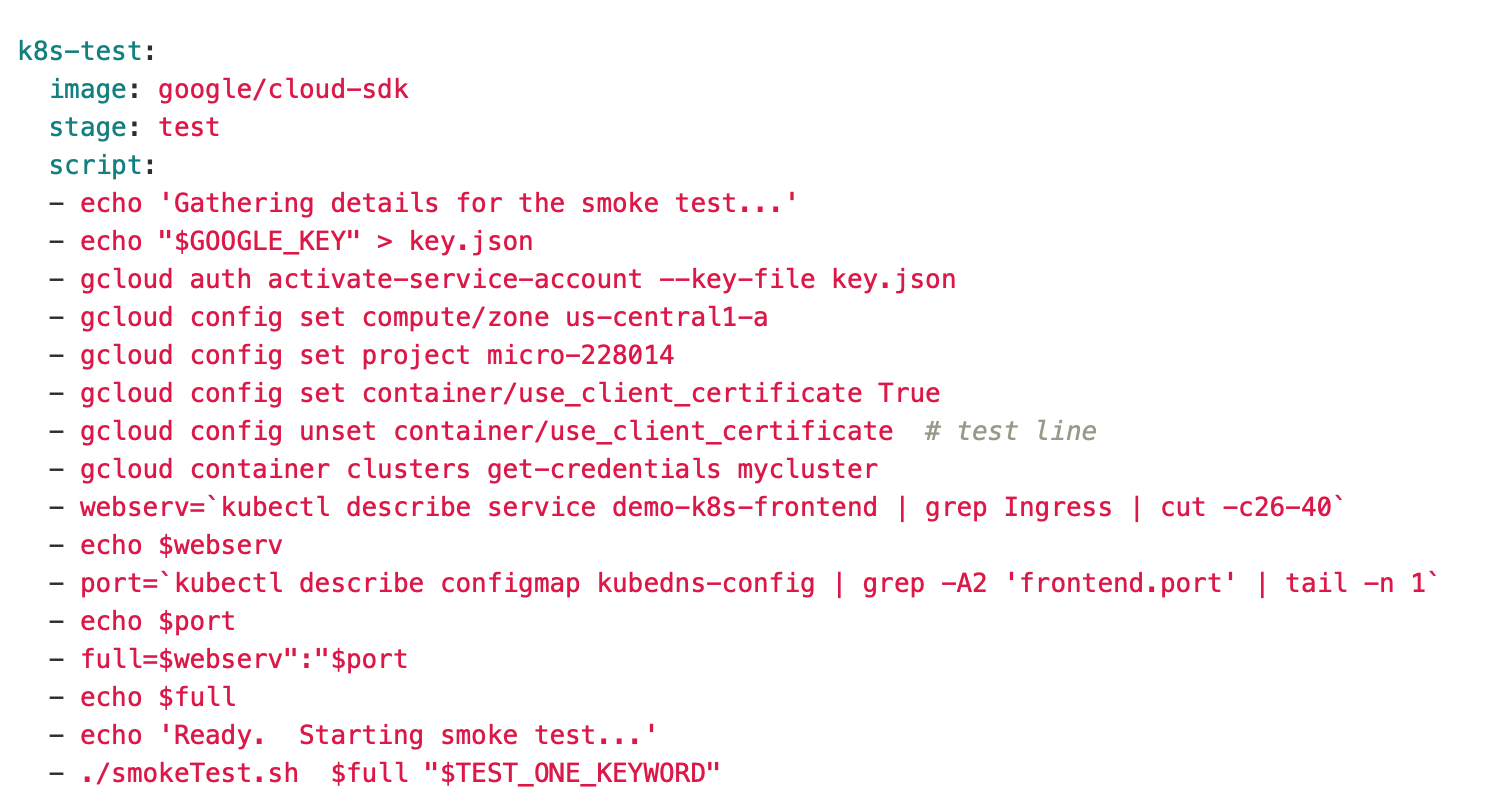
Note how we are making use of the environment variable GOOGLE\_KEY in order to authorize the pipeline actions. This step may take a few minutes while the nodes spin up.

Now we have a running cluster with 3 nodes. The next step deploys our microservice Java Spring application container to the cluster.



Once this succeeds, we have our microservice container deployed to the pod(s) in our cluster. The definition of those pods is based on deployment files, which we look at in detail below.

Now we are ready to test the running application. A couple things of note in this step are that we need an IP and port for running a REST test against the application. Instead of hard coding those values into the script (and Java code) we are using a concept of Kubernetes called a Deployment and [ConfigMap](https://kubernetes.io/docs/tasks/configure-pod-container/configure-pod-configmap/) to store them. We use standard Google Cloud console linux commands to describe those objects, and then some standard linux tools to parse out the needed information.



The entire point of the test script is to gather the IP and Port details by inspecting Kubernetes, and then using those values as parameters to a simple smoke test command file. The smoke test executes a curl command against our web application, and looks for a specific string in the output.

Here is the smokeTest file.



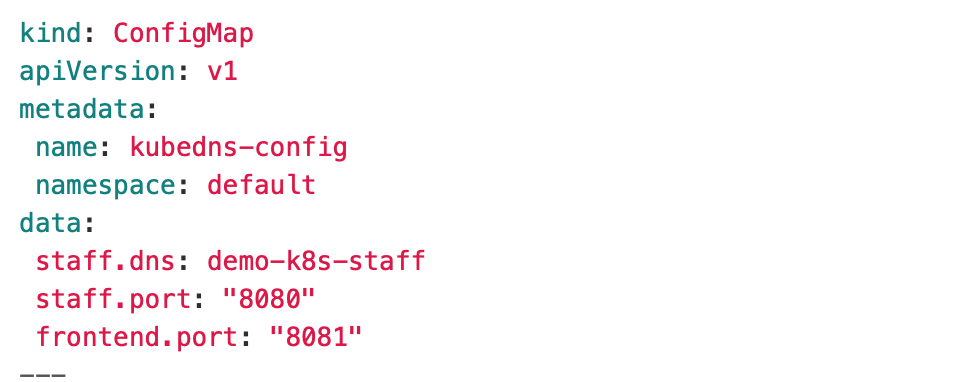
Finally, now that our pipeline has built, containerized, deployed and tested (live) our microservice application, we want to reduce our cluster size to 0 so as to avoid incurring additional costs.



Note that this is only the pipeline for deploying a single service. The code in this example includes a pair of services (and containers) that get built and deployed in their own respective repositories. i.e., we only build the code for a microservice when that particular microservice has changed.

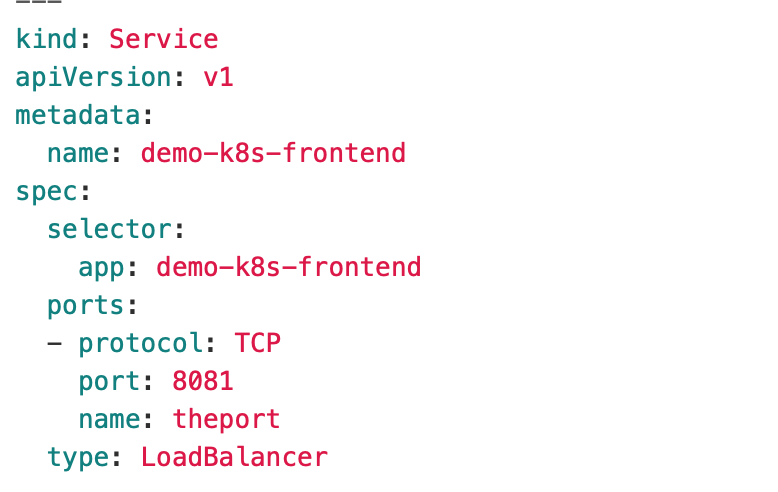
**The Kubernetes Components**

Now, let’s take a look at the Kubernetes objects. There are 3 components to our Kubernetes application, (a) a ***ConfigMap*** for storing basic variables, (b) a ***Service***, for enabling ingress to our application from the outside world, and (c) a ***deployment*** specifying the number of replicas (pods) that we want to deploy for our microservice. Here are the files.



Note that we used some of this information previously in our pipeline. We will also use in our Java Spring application, so that it is not hardcoded anywhere.

Next, the Service. We are deploying a LoadBalancer type service on port 8081, which will provide us with an externally accessible IP. This is not the least expensive way to access a pod, but for purposes of this demonstration it is the simplest. Look into an [Ingress](https://kubernetes.io/docs/concepts/services-networking/ingress/) service for a less expensive approach.



And, the deployment of the pods themselves. Note that the pod specification includes a reference to the ConfigMap elements defined earlier, which makes that information globally available to the code running in the pod (container). Again, note that the registry details for gitlab are for my own project; your project will have your own details.



**The Microservices Java Application (Spring)**

All of our work up till now has assumed we have an application to containerize and deploy to Kubernetes. Now, let’s look at the application itself. It’s a standard Spring Boot web application which represents a front-end to a 2-tier application (ie., having a backend container to operate against).

We’ll just look at the Controller class. The rest of it is quite standard.



Our application is simple 2 microservices which communicate with each other, the first being one accessed by the outside world (via the LoadBalancer IP), and the second being that which is called into from the first one. We use Kube DNS to discover the second container.

What we want to especially observe here is that we retrieved our value for the URL related to our backend service via the ConfigMap, and also the port for that service. Thus, no hardcoding in the java files, or copying values in multiple places. The ConfigMap in our deployment file provided these values as global environment variables to any container running in that pod’s deployment.

**Conclusion**

In this article we saw how to build a Gitlab.com pipeline for continuous delivery of a Java Spring microservices application into a Google Cloud Kubernetes engine.