**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. 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

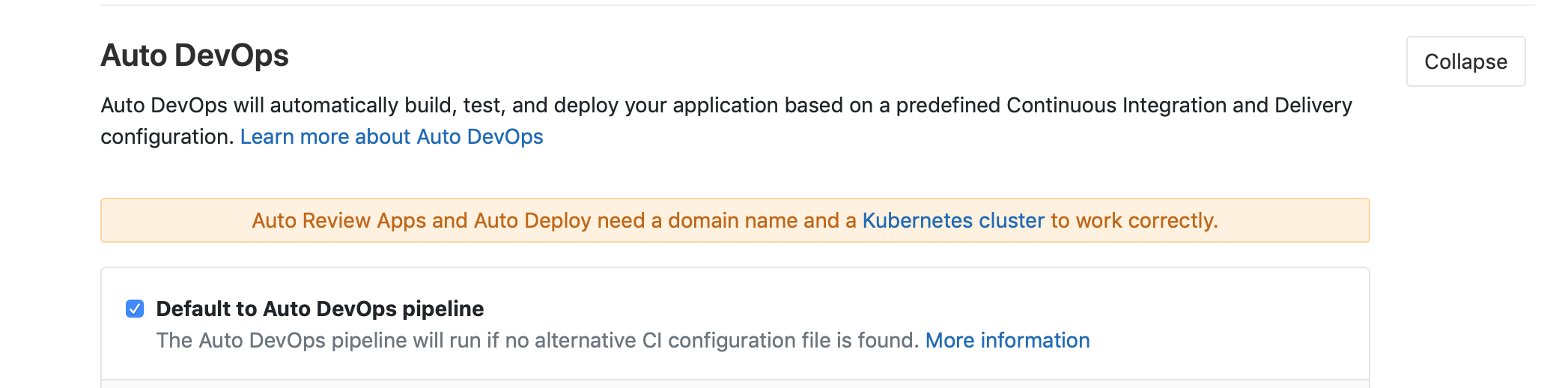
git add .

git commit -m "Initial commit"

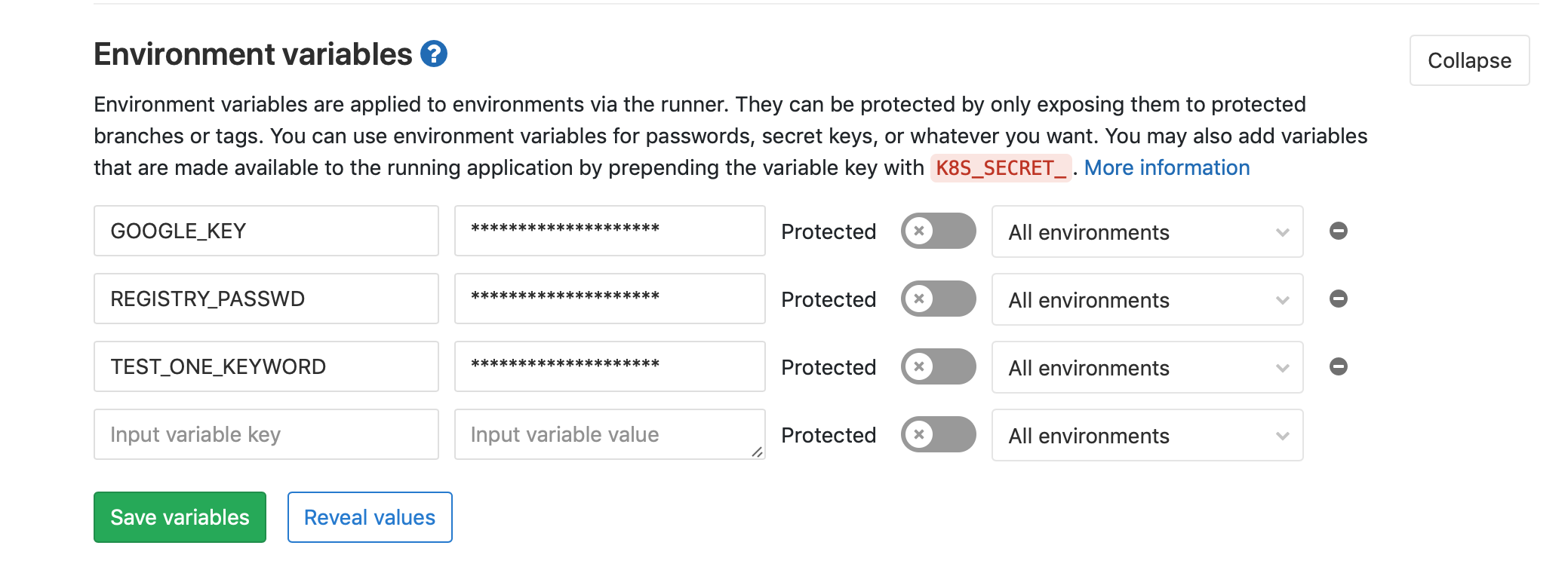
git push -u origin master

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

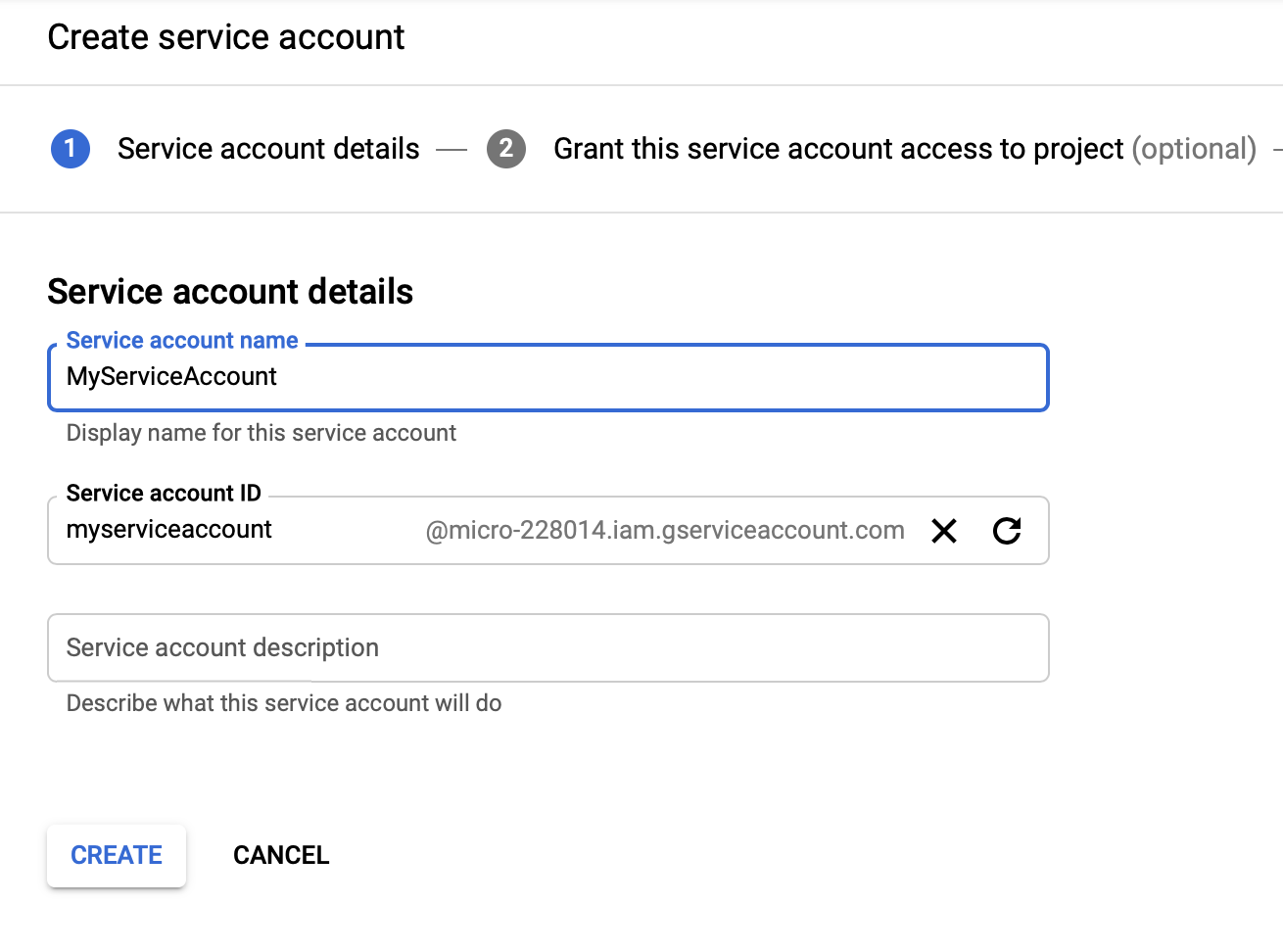
1. Turn on Auto-Devops. In your Gitlab project, go to Settings 🡪 CI/CD and expand out the Auto DevOps section. Check the box as shown. Now, anytime we have a pipeline available, it will be run on each checkin to the repository.



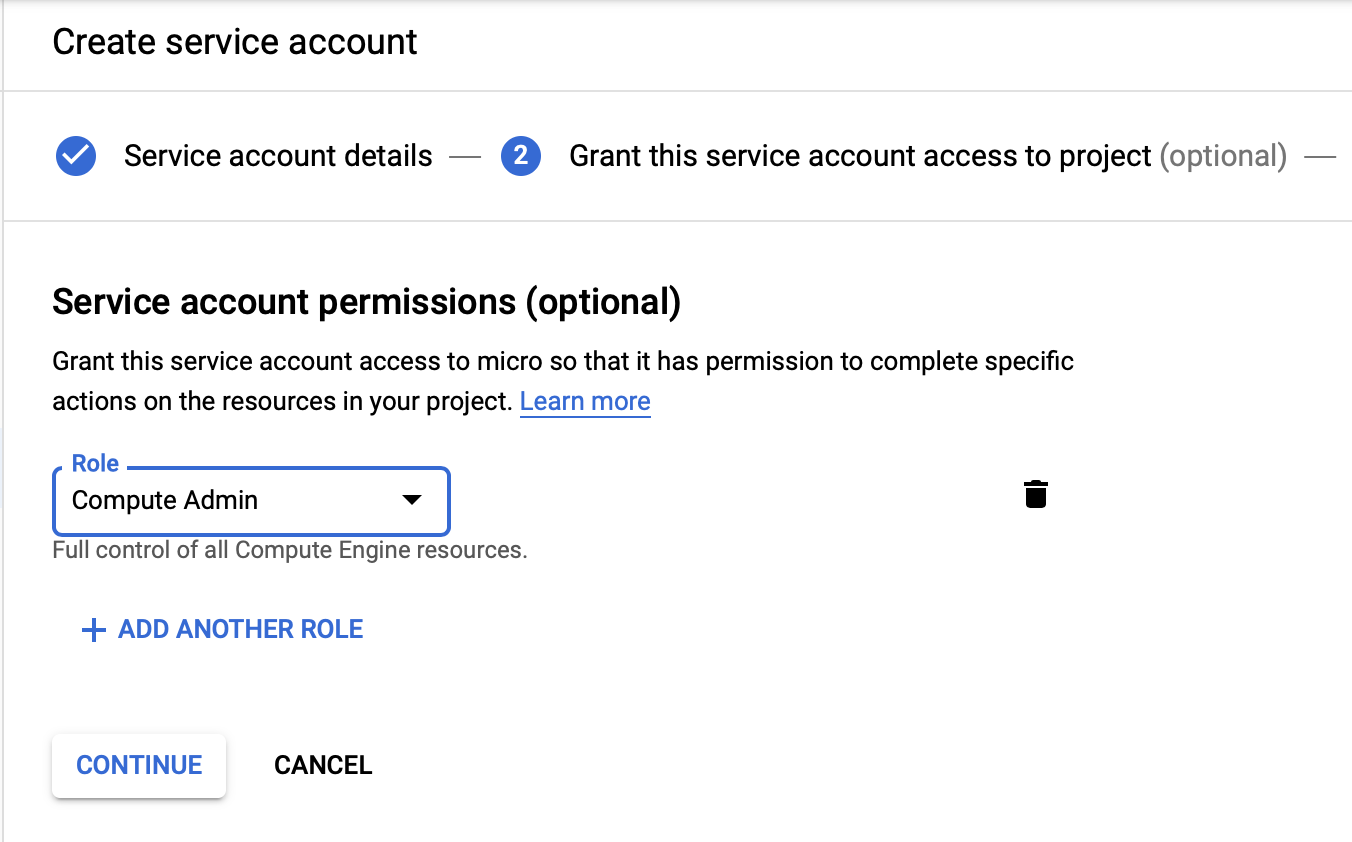
1. Setup some basic environment variables that will be needed for our pipeline to run. We’ll need 3 variables to be stored here, (a) a Google Key to enable the pipeline in Gitlab to execute commands on our Kubernetes cluster on our behalf, (b) a password to the Gitlab repository where we will store the 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.



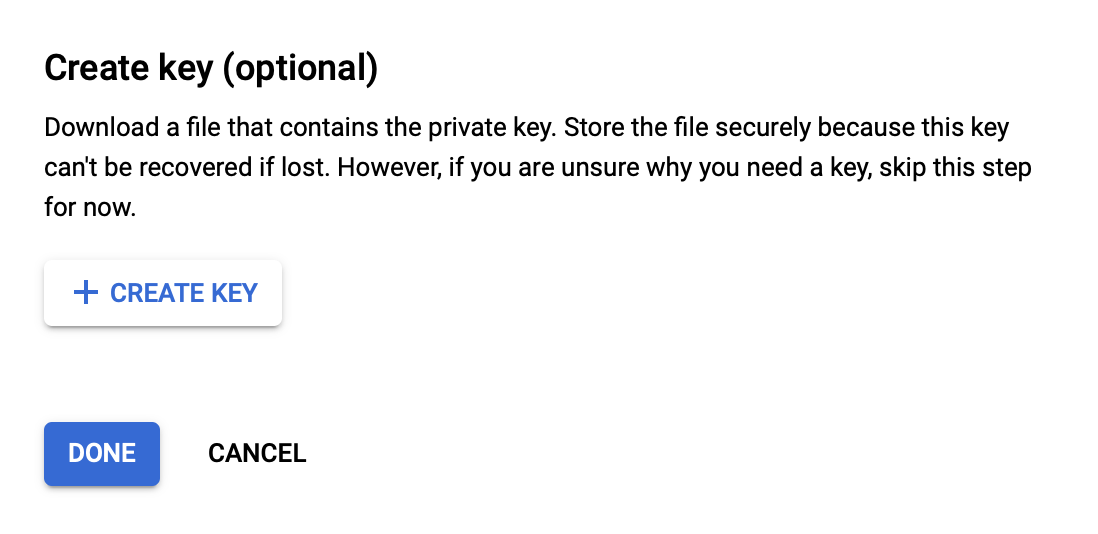
1. **The Google Key**. This is the key representing a service account in Google Cloud Platform which is authorized to do specific actions relative to the Kubernetes cluster. First, go to your cloud console for the project you want to use, and select IAM \* Admin 🡪 Service Account. Click Create Service Account.



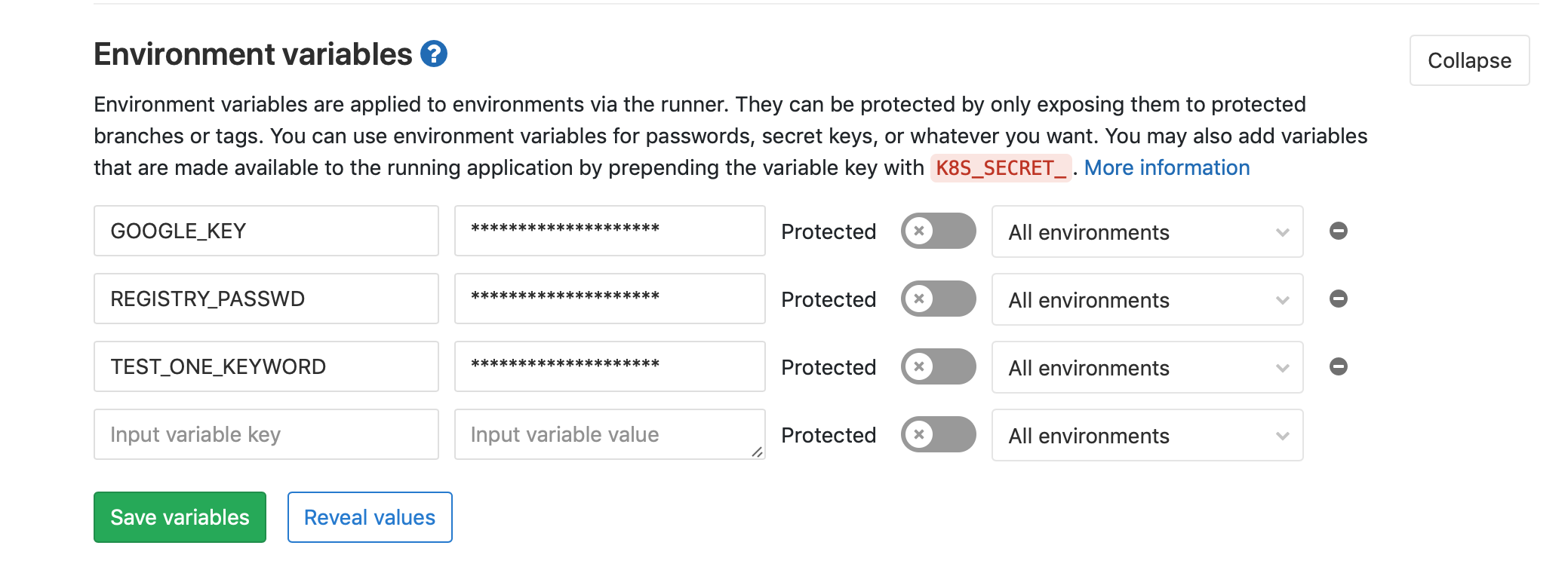
Give this account some permissions. We will give it Compute Engine admin, so it can update our cluster as part of the pipeline.



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



Now we can add it to our Gitlab environment variables. Create a new variable called GOOGLE\_KEY and then simply paste in the entire contents of the JSON file into the box next to it. Click Save.



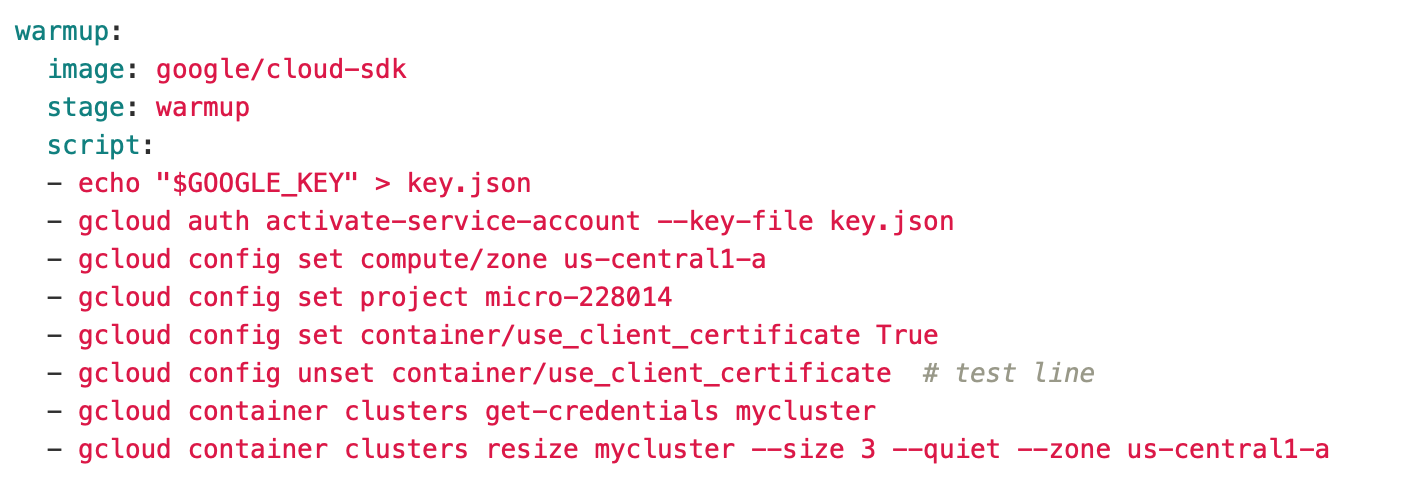
**The Pipeline**

Let’s take a look at the pipeline now. The 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.



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.

First, the warmup stage. Here, we are scaling up our deployment and testing cluster. It is kept at 0 nodes when not in use, to save resource cost. This stage will use a Google Cloud console container, 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.

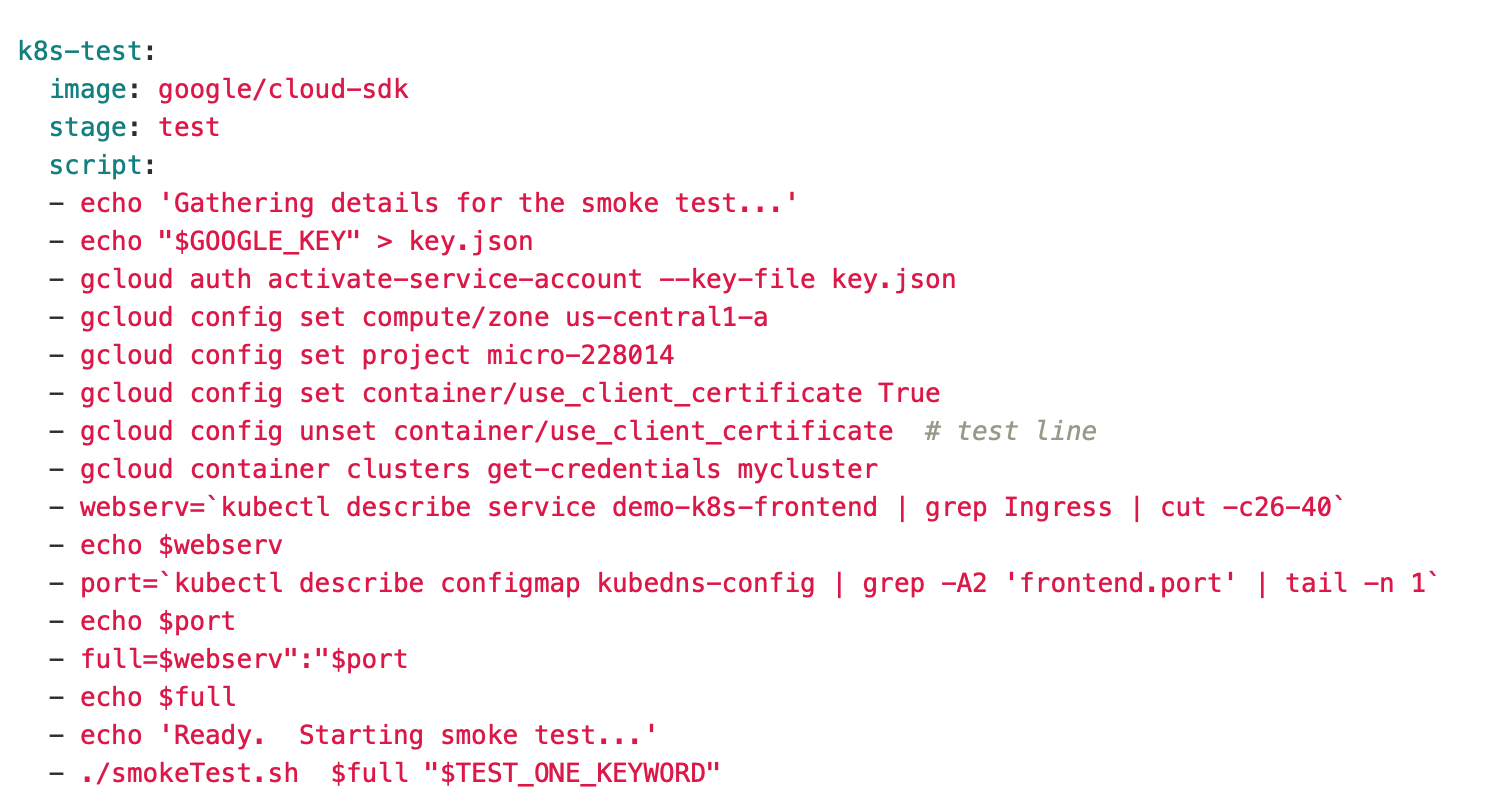
The next 2 stages build, package (containerize) and then store the application on Gitlab’s repository.



The next step deploys our container to the Google Cloud Kubernetes engine. Be sure to update the script to include your gitlab name and project.



Next, we test. The earlier Maven build would have run our Unit tests, but we want to run an actual integration test on our checkin to further validate the changes. Note that we are not hard coding the URL or the port related to our Kubernetes pod and service (explained below), but get them programmatically.



Here is the smokeTest file. Very simple check for a REST url.



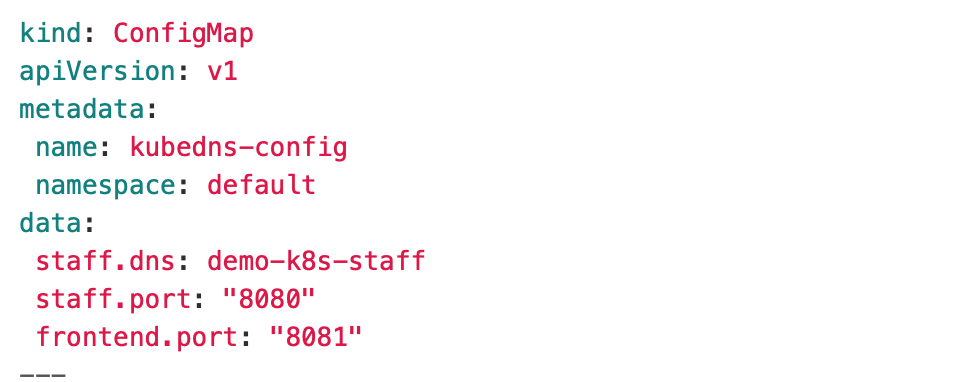
Finally, now that our pipeline has built, containerized, deployed and tested (live) our microservice application, we want to reduce our cluster size to 0 so as to avoid incurring additional costs.



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.

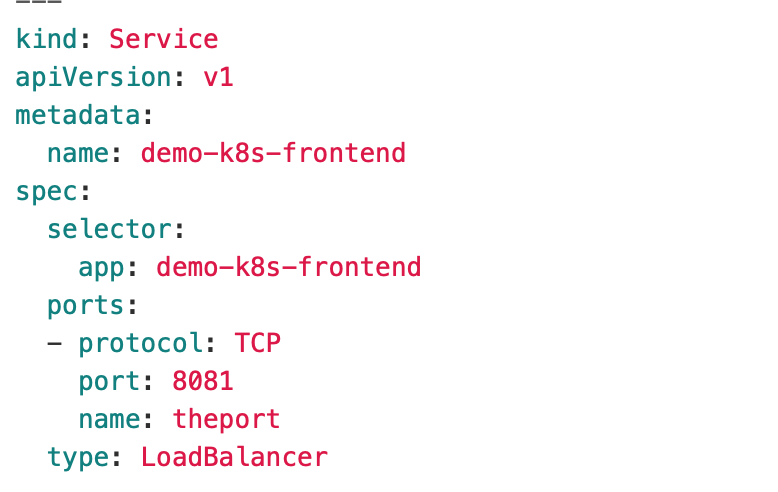
**The Kubernetes Components**

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.



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 that work up till now has assumed we have an application to containerize and deploy to Kubernetes. Now, let’s look at the application itself.

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.

**Conclusion**

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