Kubernetes Basics v1.6

3 hours1 Credit

Rate Lab

Overview

In this lab, you learn how to:

- Provision a Kubernetes cluster using Google Kubernetes Engine.
- Deploy and manage Docker containers using `kubectl`.
- Split an application into microservices using Kubernetes' Deployments and Services.
  You use Kubernetes Engine and its Kubernetes API to deploy, manage, and upgrade applications. You use an example application called "app" to complete the labs.

App is hosted on GitHub. It's a 12-Factor application with the following Docker images:

- **Monolith**: includes auth and hello services.
- **Auth** microservice: generates JWT tokens for authenticated users.
- **Hello** microservice: greets authenticated users.
- nginx: frontend to the auth and hello services.

Setup

Step 1

Before you click the Start Lab button

Read these instructions. Labs are timed and you cannot pause them. The timer, which starts when you click Start Lab, shows how long Cloud resources will be made available to you.

This Qwiklabs hands-on lab lets you do the lab activities yourself in a real cloud environment, not in a simulation or demo environment. It does so by giving you new, temporary credentials that you use to sign in and access the Google Cloud Platform for the duration of the lab.

What you need

To complete this lab, you need:

- Access to a standard internet browser (Chrome browser recommended).
- Time to complete the lab.
  Note: If you already have your own personal GCP account or project, do not use it for this lab.

Step 2

Make sure the following APIs are enabled in Cloud Platform Console:

- Kubernetes Engine API
- Container Registry API
  On the **Navigation menu** (☰), click **APIs & services**.
  Scroll down and confirm that your APIs are enabled.

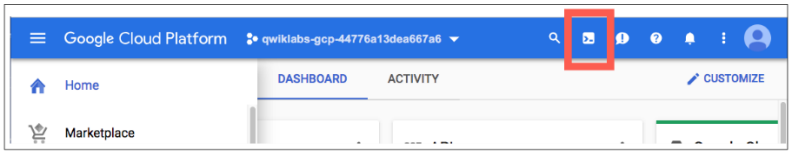
If an API is missing, click **ENABLE APIS AND SERVICES** at the top, search for the API by name, and enable it for your project.

Step 3

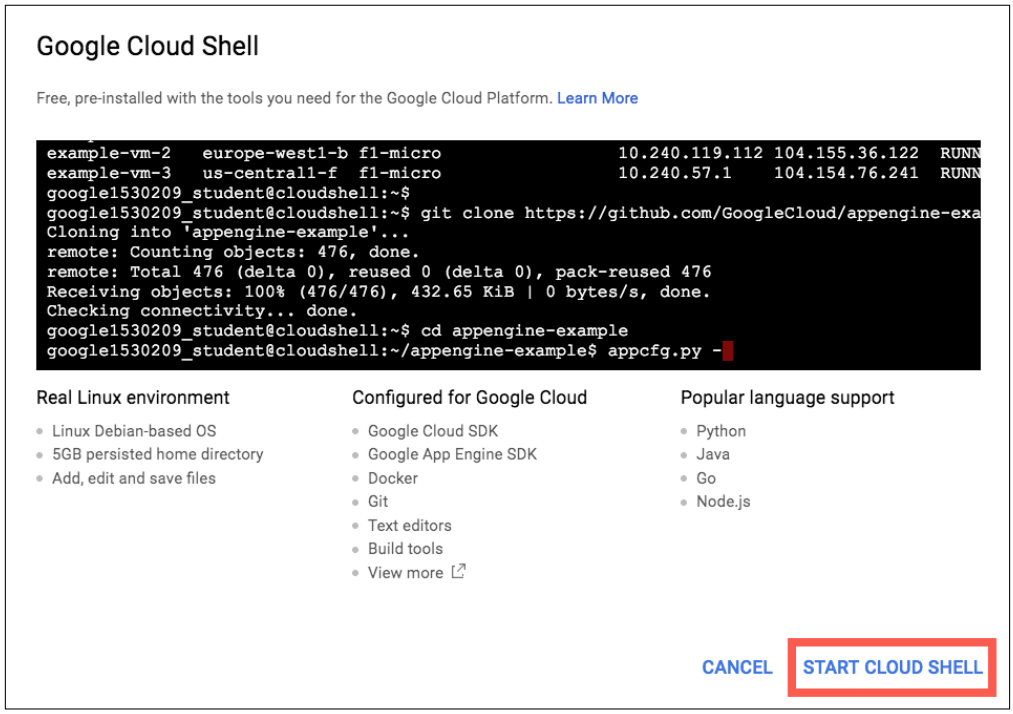
Activate Google Cloud Shell

Google Cloud Shell is a virtual machine that is loaded with development tools. It offers a persistent 5GB home directory and runs on the Google Cloud. Google Cloud Shell provides command-line access to your GCP resources.

1. In GCP console, on the top right toolbar, click the Open Cloud Shell button.

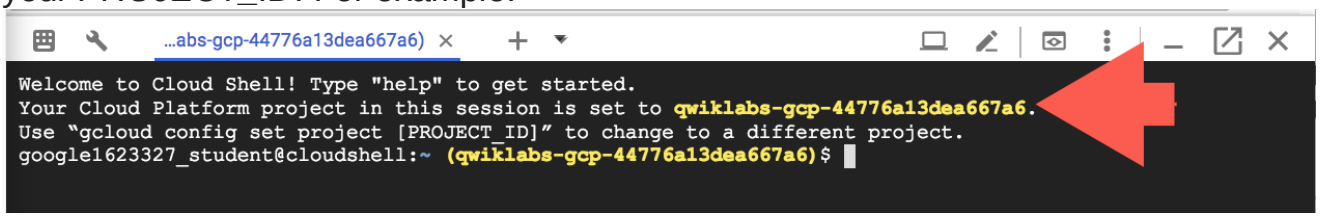


2. In the dialog box that opens, click **START CLOUD SHELL**:



You can click "START CLOUD SHELL" immediately when the dialog box opens.

It takes a few moments to provision and connect to the environment. When you are connected, you are already authenticated, and the project is set to your *PROJECT_ID*. For example:



gcloud is the command-line tool for Google Cloud Platform. It comes pre-installed on Cloud Shell and supports tab-completion.
You can list the active account name with this command:

```
gcloud auth list
```

Output:

```
Credentialed accounts:
 - <myaccount>@<mydomain>.com (active)
```

Example output:

```
Credentialed accounts:
 - google1623327_student@qwiklabs.net
```

You can list the project ID with this command:

```
gcloud config list project
```

Output:

```
[core]
project = <project_ID>
```

Example output:

```
[core]
project = qwiklabs-gcp-44776a13dea667a6
```

Full documentation of **gcloud** is available on Google Cloud gcloud Overview.

Step 4

Get the sample code from the Git repository.

```
git clone https://github.com/googlecodelabs/orchestrate-with-kubernetes.git
```

Step 5

Review the app layout.

```
cd orchestrate-with-kubernetes/kubernetes

ls
```

You'll see the following structure.

deployments/	**Deployment manifests**
nginx/	**nginx config files**

pods/	**pod manifests**
services/	**Services manifests**
tls/	**TLS certificates**
cleanup.sh	**Cleanup script**

Now that you have the code, it's time to try Kubernetes.

A quick demo of Kubernetes

Start a Kubernetes cluster

Step 1

Define your zone as a project default zone. This way you do not need to specify --zone parameter in gcloud commands.

```
gcloud config set compute/zone us-central1-a
```

In Cloud Shell, run the following command to start a Kubernetes cluster called `bootcamp` that runs 5 nodes.

```
gcloud container clusters create bootcamp --num-nodes 5 --scopes "https://www.googleapis.com/auth/projecthosting,storage-rw"
```

The `scopes` argument provides access to project hosting and Google Cloud Storage APIs that you'll use later.
It takes several minutes to create a cluster as Kubernetes Engine provisions virtual machines for you. It spins up one or more master nodes and multiple configured worker nodes. This is one of the advantages of a managed service.
Click *Check my progress* to verify the objective.

Create a Kubernetes cluster

Check my progress

Step 2

After the cluster is created, check your installed version of Kubernetes using the `kubectl version` command.

```
kubectl version
```

The `gcloud container clusters create` command automatically authenticated `kubectl` for you.

Step 3

Use `kubectl cluster-info` to find out more about the cluster.

```
kubectl cluster-info
```

Step 4

View your running nodes in Cloud Platform Console.

Open the **Navigation menu** and go to **Compute Engine > VM Instances**.
Congratulations! Your Kubernetes cluster is now ready for use!

Bash Completion (Optional)

Kubernetes comes with auto-completion. You can use the kubectl completion command and the built-in `source` command to set this up.

Step 1

Run this command.

```
source <(kubectl completion bash)
```

Step 2

Press **Tab** to display a list of available commands.
Try the following examples:

```
kubectl <TAB><TAB>
annotate        autoscale       convert         describe        expose
api-versions    certificate     cordon          drain           get
apply           cluster-info    cp              edit            label
```

You can also complete a partial command.

```
kubectl co<TAB><TAB>
completion  config      convert     cordon
```

This feature makes using `kubectl` even easier.

Run and deploy a container

The easiest way to get started with Kubernetes is to use the `kubectl run` command.

Step 1

Use `kubectl run` to launch a single instance of the nginx container.

```
kubectl run nginx --image=nginx:1.10.0
```

In Kubernetes, all containers run in pods. And in this command, Kubernetes created what is called a `deployment` behind the scenes, and runs a single pod with the nginx container in it. A deployment keeps a given number of pods up and running even when the nodes they run on fail. In this case, you run the default number of pods, which is 1.
You'll learn more about deployments later.

Step 2

Use the `kubectl get pods` command to view the pod running the nginx container.

```
kubectl get pods
```

Step 3

Use the `kubectl expose` command to expose the nginx container outside Kubernetes.

```
kubectl expose deployment nginx --port 80 --type LoadBalancer
```

Kubernetes created a `service` and an external load balancer with a public IP address attached to it (you will learn about services later). The IP address remains the same for the life of the service. Any client who hits that public IP address (for example an end user or another container) is routed to pods behind the service. In this case, that would be the nginx pod.

Step 4

Use the `kubectl get` command to view the new service.

```
kubectl get services
```

You'll see an external IP that you can use to test and contact the nginx container remotely.

It may take a few seconds before the `ExternalIP` field is populated for your service. This is normal—just re-run the `kubectl get services` command every few seconds until the field is populated.

Step 5

Use the `kubectl scale` command to scale up the number of backend applications (*pods*) running on your service using.

```
kubectl scale deployment nginx --replicas 3
```

This is useful when you want to increase workload for a web application that is becoming more popular.

Step 6

Get the pods one more time to confirm that Kubernetes has updated the number of pods.

```
kubectl get pods
```

Step 7

Use the `kubectl get services` command again to confirm that your external IP address has not changed.

```
kubectl get services
```

Step 8

Use the external IP address with the `curl` command to test your demo application.

```
curl http://<External IP>:80
```

Kubernetes supports an easy-to-use workflow out of the box using the `kubectl run`, `expose`, and `scale` commands.

Clean Up

Clean up nginx by running the following commands.

```
kubectl delete deployment nginx
kubectl delete service nginx
```

Now that you've seen a quick tour of Kubernetes, it's time to dive into each of the components and abstractions.

You covered a lot of information. The rest of this lab goes over these concepts in depth. You can always come back to this demo if you need to see it again.

Pods

Investigate pods in more detail.

Creating Pods

Pods can be created using pod configuration files.

Step 1

Explore the built-in pod documentation using the `kubectl explain` command.

```
kubectl explain pods
```

While you explore the Kubernetes API, `kubectl explain` will be one of the most common commands you use. Note how you used it above to investigate an API object and how you will use it below to check on various properties of API objects.

Step 2

Explore the monolith pod's configuration file.

```
cat pods/monolith.yaml
```

The pod is made up of one container (called `monolith`). You pass a few arguments to the container when it starts up and open port 80 for HTTP traffic.

Step 3

Use the `kubectl explain` command with the `.spec` option to view more information about API objects. This example inspects containers.

```
kubectl explain pods.spec.containers
```

Explore the rest of the API before you continue.

Step 4

Create the `monolith` pod using `kubectl create`.

```
kubectl create -f pods/monolith.yaml
```

Click *Check my progress* to verify the objective.

Create the monolith pod

Check my progress

Step 5

Use the `kubectl get pods` command to list all pods running in the default namespace.

```
kubectl get pods
```

It may take a few seconds before the monolith pod is up and running, because the monolith container image must be pulled from the Docker Hub before you can run it.

Step 6

When the pod is running, use the `kubectl describe` command to get more information about the `monolith` pod.

```
kubectl describe pods monolith
```

You'll see a lot of the information about the `monolith` pod, including the pod IP address and the event log. This information will be useful when troubleshooting.
It's time for a quick knowledge check. Answer the following questions about the monolith pod.

- What is the pod IP address?
- Which node is the pod running on?
- Which containers are running in the pod?
- Which labels are attached to the pod?
- Which arguments are set on the container?

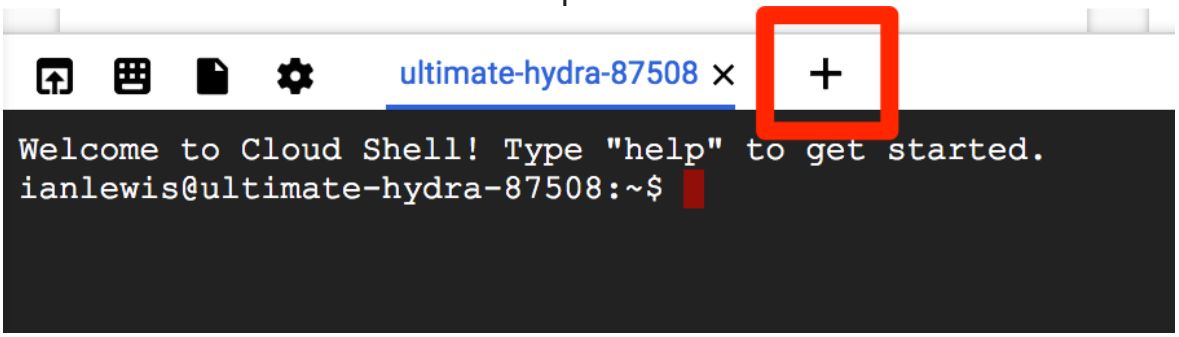
As you can see, Kubernetes makes it easy to create pods by describing them in configuration files and to view information about them when they are running. At this point, you can create all the pods your deployment requires!

Interacting with pods

Pods are allocated a private IP address by default that cannot be reached outside of the cluster. Use the `kubectl port-forward` command to map a local port to a port inside the `monolith` pod.
Use two terminals: one to run the `kubectl port-forward` command, and the other to issue `curl` commands.

Step 1

Click the `+` button in Cloud Shell to open a new terminal.



Step 2

Run the following command to set up port-forwarding from a local port, 10080, to a pod port, 80 (where your container is listening).

```
kubectl port-forward monolith 10080:80
```

Step 3

To access your pod, return to the first terminal window and run the following `curl` command.

```
curl http://127.0.0.1:10080
```

You get a friendly "Hello" back from the container.

Step 4

See what happens when you hit a secure endpoint.

```
curl http://127.0.0.1:10080/secure
```

You should get an error.

You get an error because you need to include an auth token in your request.

Step 5

Log in to get an auth token from `monolith`.

```
curl -u user http://127.0.0.1:10080/login
```

At the login prompt, enter the password as `password` to sign in.
Logging in causes a JWT token to be printed out. You'll use it to test your secure endpoint with `curl`.

Step 6

Cloud Shell doesn't handle copying long strings well, so copy the token into an environment variable.

```
TOKEN=$(curl http://127.0.0.1:10080/login -u user|jq -r '.token')
```

At the login prompt, enter the password as `password` to sign in.

Step 7

Access the secure endpoint again, and this time include the auth token.

```
curl -H "Authorization: Bearer $TOKEN" http://127.0.0.1:10080/secure
```

You should get a response back from your application letting you know it works again!

Step 8

Use the `kubectl logs` command to view logs for the `monolith` pod.

```
kubectl logs monolith
```

Step 9

Open another terminal and use the `-f` flag to get a stream of logs in real-time!
To create the third terminal, click the `+` button in Cloud Shell and run the following command.

```
kubectl logs -f monolith
```

Step 10

Use `curl` in terminal 1 to interact with `monolith`. And you see logs update in terminal 3.

```
curl http://127.0.0.1:10080
```

You can see the logs updating back in terminal 3.

Step 11

Use the `kubectl exec` command to run an interactive shell inside the `monolith` pod. This can be useful when you want to troubleshoot from within a container.

```
kubectl exec monolith --stdin --tty -c monolith /bin/sh
```

Step 12

Optional: In the shell, you can test external (outward facing) connectivity using the `ping` command.

```
ping -c 3 google.com
```

Step 13

Sign out of the shell.

```
exit
```

As you can see, interacting with pods is as easy as using the `kubectl` command. If you need to test a container remotely or get a login shell, Kubernetes provides everything you need to start.

Step 14

To quit `kubectl port-forward` and `kubectl logs` in terminal 2 and 3, press `Ctrl+C`.

Monitoring and Health Checks

Kubernetes supports monitoring applications in the form of readiness and liveness probes. Health checks can be performed on each container in a pod. Readiness probes indicate when a pod is "ready" to serve traffic. Liveness probes indicate whether a container is "alive." If a liveness probe fails multiple times, the container is restarted. Liveness probes that continue to fail cause a pod to enter a crash loop. If a readiness check fails, the container is marked as *not ready* and is removed from any load balancers.
In this lab, you deploy a new pod named `healthy-monolith`, which is largely based on the `monolith` pod with the addition of readiness and liveness probes.
In this lab, you learn how to:

- Create pods with readiness and liveness probes.
- Troubleshoot failing readiness and liveness probes.

Creating Pods with Liveness and Readiness Probes

Step 1

Explore the `healthy-monolith` pod configuration file.

```
cat pods/healthy-monolith.yaml
```

Step 2

Create the `healthy-monolith` pod using `kubectl`.

```
kubectl create -f pods/healthy-monolith.yaml
```

Click *Check my progress* to verify the objective.

Create the healthy-monolith pod

Check my progress

Step 3

Pods are not marked *ready* until the readiness probe returns an HTTP 200 response. Use the `kubectl describe` command to view details for the healthy-monolith pod.

```
kubectl describe pod healthy-monolith
```

Readiness Probes

See how Kubernetes responds to failed readiness probes. The `monolith` container supports the ability to force failures of its readiness and liveness probes. This enables you to simulate failures for the `healthy-monolith` pod.

Step 1

Use the `kubectl port-forward` command in terminal 2 to forward a local port to the health port of the `healthy-monolith` pod.

```
kubectl port-forward healthy-monolith 10081:81
```

Step 2

Force the `monolith` container readiness probe to fail. Use the `curl` command in terminal 1 to toggle the readiness probe status. Note that this command does not show any output.

```
curl http://127.0.0.1:10081/readiness/status
```

Step 3

Get the status of the `healthy-monolith` pod using the `kubectl get pods -w` command.

```
kubectl get pods healthy-monolith -w
```

Step 4

Press `Ctrl+C` when there are `0/1` ready containers. Use the `kubectl describe` command to get more details about the failing readiness probe.

```
kubectl describe pods healthy-monolith
```

Step 5

Notice the events for the `healthy-monolith` pod report details about failing readiness probes.

To force the `monolith` container readiness probe to pass, toggle the readiness probe status by using the `curl` command.

```
curl http://127.0.0.1:10081/readiness/status
```

Step 6

Wait about 15 seconds and get the status of the `healthy-monolith` pod using the `kubectl get pods` command.

```
kubectl get pods healthy-monolith
```

Step 7

Press `Ctrl+C` in terminal 2 to close the `kubectl` proxy (i.e `port-forward`) command.

Liveness Probes

Building on what you learned in the previous tutorial, use the `kubectl port-forward` and `curl` commands to force the `monolith` container liveness probe to fail. Observe how Kubernetes responds to failing liveness probes.

Step 1

Use the `kubectl port-forward` command to forward a local port to the health port of the `healthy-monolith` pod in terminal 2.

```
kubectl port-forward healthy-monolith 10081:81
```

Step 2

To force the `monolith` container readiness probe to pass, toggle the readiness probe status by using the `curl` command in another terminal.

```
curl http://127.0.0.1:10081/healthz/status
```

Step 3

Get the status of the `healthy-monolith` pod using the `kubectl get pods -w` command.

```
kubectl get pods healthy-monolith -w
```

Step 4

When a liveness probe fails, the container is restarted. Once restarted, the `healthy-monolith` pod should return to a healthy state. Press `Ctrl+C` to exit that command when the pod restarts. Note the restart count.

Step 5

Use the `kubectl describe` command to get more details about the failing liveness probe. You can see the related events for when the liveness probe failed and the pod was restarted.

```
kubectl describe pods healthy-monolith
```

Step 6

When you are finished, press `Ctrl+C` in terminal 2 to close the `kubectl proxy` command.

Congratulations!

You learned about Kubernetes pods and Kubernetes support for application monitoring using liveness and readiness probes. You also learned how to add readiness and liveness probes to pods and what happens when probes fail.

Services

Next steps:

- Create a service.
- Use label selectors to expose a limited set of pods externally.

Creating a Service

Before creating your services, create a secure pod with an nginx server called `secure-monolith` that can handle HTTPS traffic.

Step 1

Create two volumes that the secure pod will use to bring in (or consume) data.

The first volume of type `secret` stores TLS cert files for your nginx server.
Return to terminal 1 and create the first volume using the following command:

```
kubectl create secret generic tls-certs --from-file tls/
```

This uploads cert files from the local directory `tls/` and stores them in a `secret` called `tls-certs`.

Create the second volume of type `ConfigMap` to hold nginx's configuration file.

```
kubectl create configmap nginx-proxy-conf --from-file nginx/proxy.conf
```

This uploads the `proxy.conf` file to the cluster and calls the ConfigMap `nginx-proxy-conf`.

Step 2

Explore the `proxy.conf` file that nginx will use.

```
cat nginx/proxy.conf
```

The file specifies that SSL is ON and specifies the location of cert files in the container file system.

The files really exist in the `secret` volume, so you need to mount the volume to the container's file system.

Step 3

Explore the `secure-monolith` pod configuration file.

```
cat pods/secure-monolith.yaml
```

Under `volumes`, the pod attaches the two volumes you created. And under `volumeMounts`, it mounts the `tls-certs` volume to the container's file system so nginx can consume the data.

Step 4

Run the following command to create the `secure-monolith` pod with its configuration data.

```
kubectl create -f pods/secure-monolith.yaml
```

Now that you have a secure pod, expose the `secure-monolith` pod externally using a Kubernetes service.

Step 5

Explore the monolith service configuration file.

```
cat services/monolith.yaml
```

The file contains:

- The `selector` that finds and exposes pods with labels `app=monolith` and `secure=enabled`
- `targetPort` and `nodePort` that forward external traffic from port 31000 to nginx on port 443.

Step 6

Use the `kubectl create` command to create the monolith service from the monolith service configuration file.

```
kubectl create -f services/monolith.yaml
```

The `type: NodePort` in the Service's yaml file means that it uses a port on each cluster node to expose the service. This means that it's possible to have port collisions if another app tries to bind to port 31000 on one of your servers.
Normally, Kubernetes handles this port assignment for you. In this lab, you chose one so that it's easier to configure health checks later.

Step 7

Use the `gcloud compute firewall-rules` command to allow traffic to the monolith service on the exposed nodeport.

```
gcloud compute firewall-rules create allow-monolith-nodeport --allow=tcp:31000
```

Now that everything is set up, you should be able to test the `secure-monolith` service from outside the cluster without using port forwarding.

Step 8

Get an IP address for one of your nodes.

```
gcloud compute instances list
```

Step 9

Try to open the URL in your browser.

```
https://<EXTERNAL_IP>:31000
```

That timed out or refused to connect. What's going wrong?
It's time for a quick knowledge check. Use the following commands to answer the questions below.

```
kubectl get services monolith
kubectl describe services monolith
```

Questions:

- Why can't you get a response from the monolith service?
- How many endpoints does the monolith service have?
- What labels must a pod have to be picked up by the monolith service?

Adding Labels to Pods

Currently the monolith service does not have any endpoints. One way to troubleshoot an issue like this is to use the `kubectl get pods` command with a label query.

Step 1

Determine that there are several pods running with the monolith label.

```
kubectl get pods -l "app=monolith"
```

Step 2

But what about `app=monolith` and `secure=enabled`?

```
kubectl get pods -l "app=monolith,secure=enabled"
```

Notice that this label query does not print any results. You need to add the "secure=enabled" label to them.

Step 3

Use the `kubectl label` command to add the missing `secure=enabled` label to the `secure-monolith` pod.

```
kubectl label pods secure-monolith 'secure=enabled'
```

Click *Check my progress* to verify the objective.

Create a secret, service, firewall rule and pod with label

Check my progress

Step 4

Check to see that your labels are updated.

```
kubectl get pods secure-monolith --show-labels
```

Step 5

View the list of endpoints on the `monolith` service.

```
kubectl get endpoints monolith
```

And you have one!

Step 6

Test this by testing one of your nodes again.

```
gcloud compute instances list | grep gke-
```

Open the following URL in your browser. You will need to click through the SSL warning because `secure-monolith` is using a self-signed certificate.

```
https://<EXTERNAL_IP>:31000
```