Inspec resource

<https://github.com/inspec/inspec-gcp/tree/master/test/integration/verify/controls>

Kitchen Terraform

<https://www.inspec.io/docs/reference/resources/#gcp-resources>

Inspec GCP examples

<https://github.com/newcontext-oss/kitchen-terraform>

Cloud Foundation Toolkit

<https://github.com/terraform-google-modules/terraform-google-project-factory>

Terraform is a popular open-source tool for defining and provisioning infrastructure (Infrastructure as Code), but few practitioners test the resulting deployments for function and security posture, in an automated way. By introducing tests and running them in the deployment pipeline, practitioners can automatically detect regressions, breaks and known security vulnerabilities early in the release process. This has the effect of increasing reliability, security and compliance of deployments in production.

In this workshop we will use Terraform with InSpec, an open-source, controls-as-code language and a remote testing engine, to deploy and test infrastructure.

What You'll Learn

In this workshop you will learn how to:

* Read and write InSpec controls and run them against a local environment
* Create an InSpec profile consisting of various controls and scan a GCP Terraform deployment
* Detect infrastructure drift with Terraform and InSpec
* Remediate infrastructure drift and confirm the remediation
* Streamline and automate the provision-deploy-test-change-deploy-test-destroy workflow with kitchen-terraform

**Setup**

**What you'll need**

To complete this lab, you'll need:

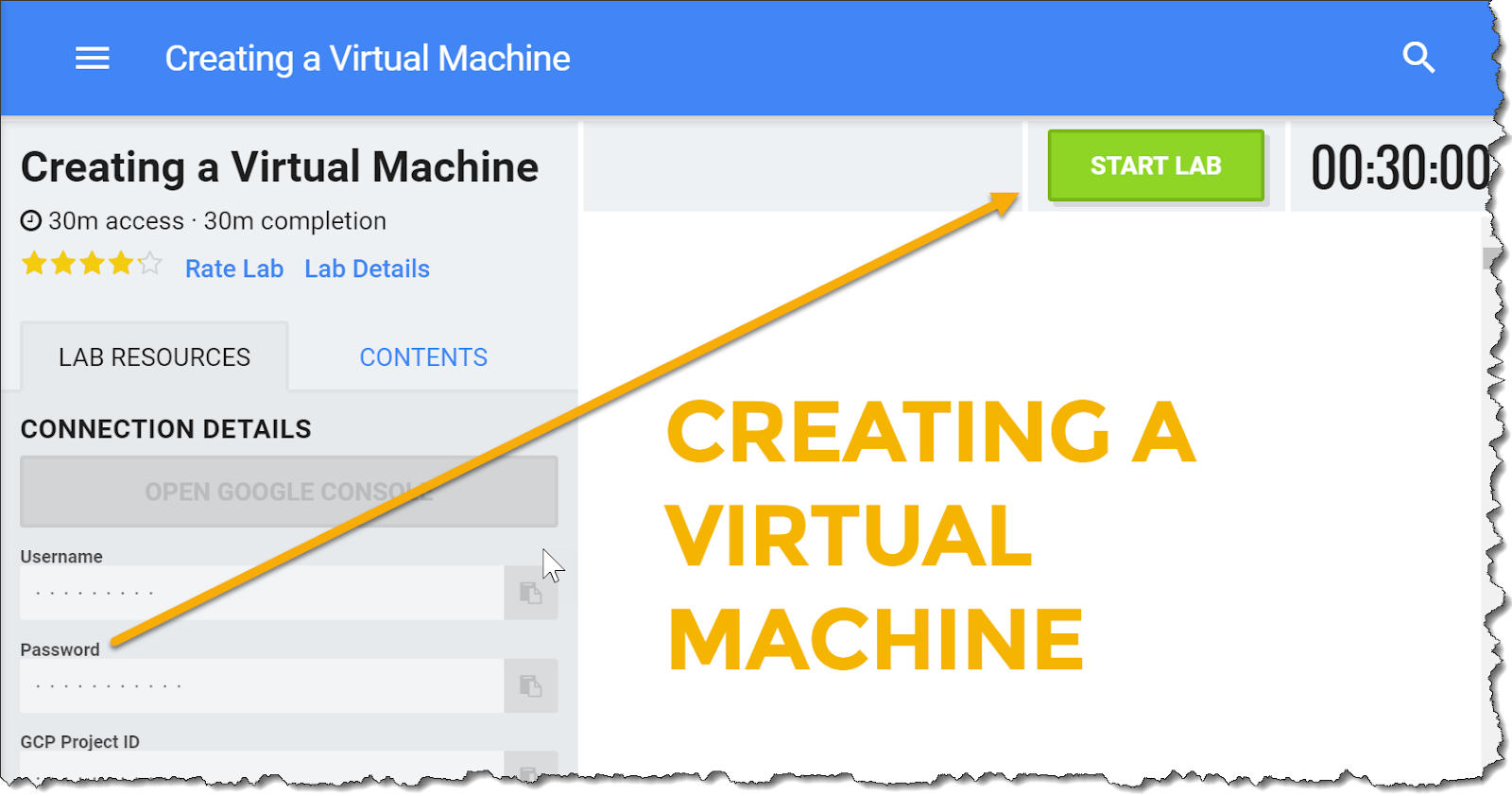
* Access to a standard internet browser (Chrome browser recommended).
* Time. Note the lab's **Completion** time in Qwiklabs, which is an estimate of the time it should take to complete all steps. Plan your schedule so you have time to complete the lab. Once you start the lab, you will not be able to pause and return later (you begin at step 1 every time you start a lab).
* You do NOT need a Google Cloud Platform account or project. An account, project and associated resources are provided to you as part of this lab.
* If you already have your own GCP account, make sure you do not use it for this lab.
* If your lab prompts you to log into the console, **use only the student account provided to you by the lab**. This prevents you from incurring charges for lab activities in your personal GCP account.

Use a new Incognito window (Chrome) or another browser for the Qwiklabs session. Alternatively, you can log out of all other Google / Gmail accounts before beginning the labs.



**Start your lab**

When you are ready, click **Start Lab**. You can track your lab's progress with the status bar at the top of your screen.

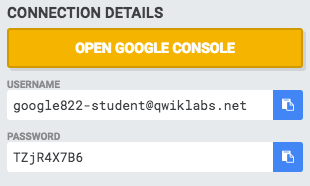


**Important**: What is happening during this time?

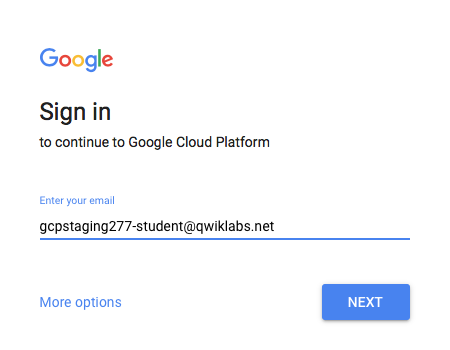
Your lab is spinning up GCP resources for you behind the scenes, including an account, a project, resources within the project, and permission for you to control the resources you will need to run the lab. This means that instead of spending time manually setting up a project and building resources from scratch as part of your lab, you can begin learning more quickly.

**Find Your Lab's GCP Username and Password**

To access the resources and console for this lab, locate the Connection Details panel in Qwiklabs. Here you will find the account ID and password for the account you will use to log in to the Google Cloud Platform:



If your lab provides other resource identifiers or connection-related information, it will appear on this panel as well.

****

**Sign in to Google Cloud Console**

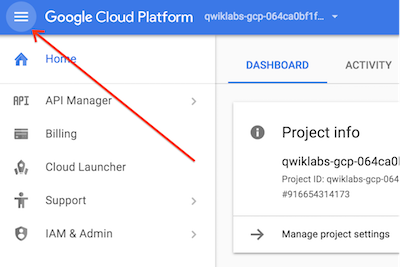
Using the Qwiklabs browser tab/window (preferably in Incognito mode) or the separate browser you are using for the Qwiklabs session, copy the Username from the Connection Details panel and click **Open Google Console**. Paste in the Username and then the Password as prompted:

Accept the terms and conditions.

Because this is a temporary account, which you will only have access to for this one lab:

* Do not add recovery options
* Do not sign up for free trials

**Note:**You can view the menu with a list of GCP Products and Services by clicking the button at the top-left next to "Google Cloud Platform."

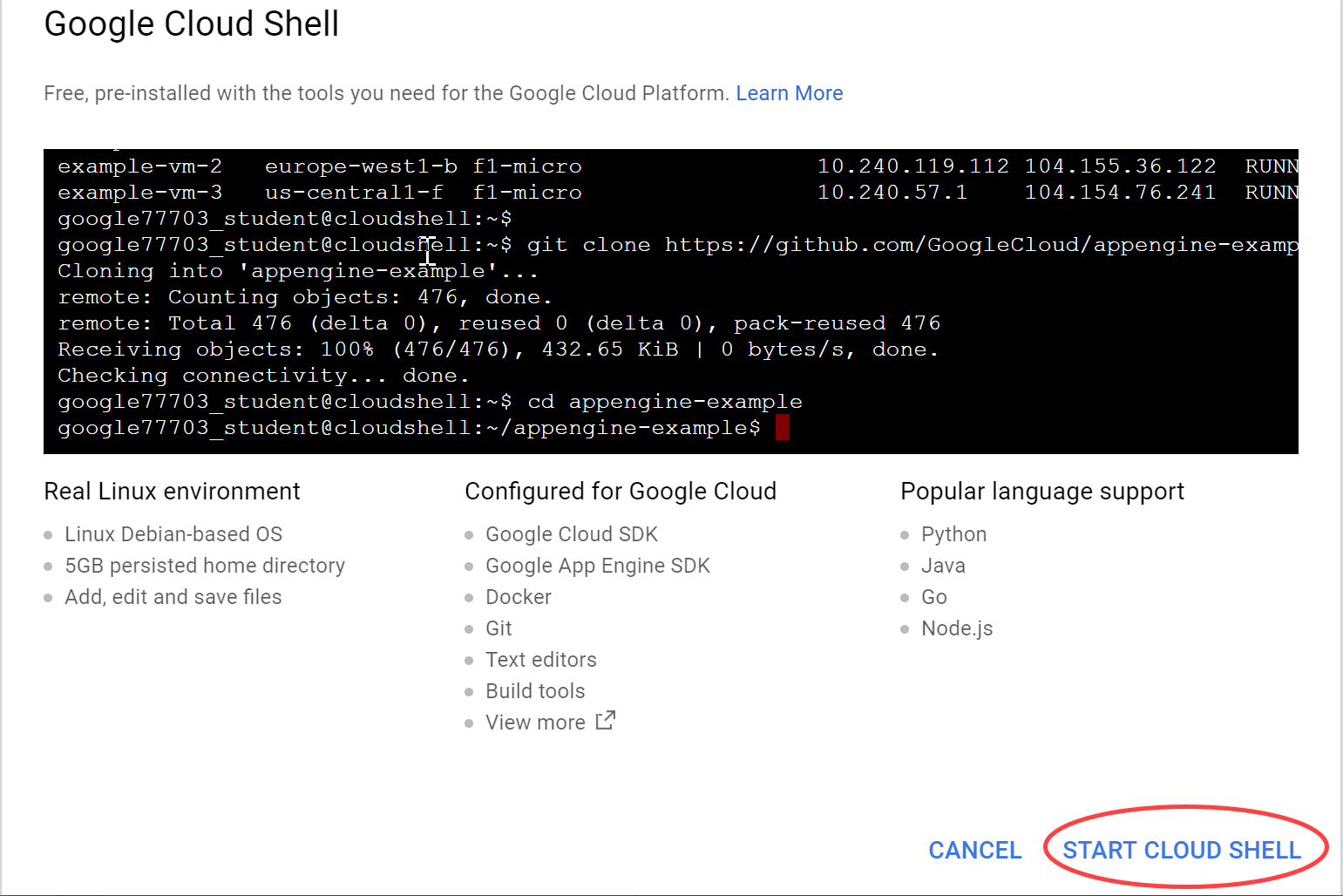


**Activate Google Cloud Shell**

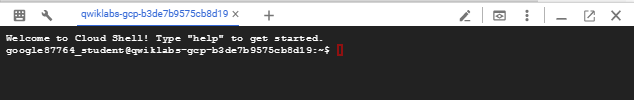
From the GCP Console click the Cloud Shell icon on the top right toolbar:



Then click "Start Cloud Shell":



It should only take a few moments to provision and connect to the environment:



This virtual machine is loaded with all the development tools you'll need. It offers a persistent 5GB home directory, and runs on the Google Cloud, greatly enhancing network performance and authentication. Much, if not all, of your work in this lab can be done with simply a browser or your Google Chromebook.

Once connected to Cloud Shell, you should see that you are already authenticated and that the project is already set to your *PROJECT\_ID*.

Run the following command in Cloud Shell to confirm that you are authenticated:

gcloud auth list

**Command output**

Credentialed accounts:

- <myaccount>@<mydomain>.com (active)

**Note:**gcloud is the powerful and unified command-line tool for Google Cloud Platform. Full documentation is available from [https://cloud.google.com/sdk/gcloud](https://cloud.google.com/sdk/gcloud/). It comes pre-installed on Cloud Shell. You will notice its support for tab-completion.

gcloud config list project

**Command output**

[core]

project = <PROJECT\_ID>

If it is not, you can set it with this command:

gcloud config set project <PROJECT\_ID>

**Command output**

Updated property [core/project].

**Lab 1: Write local tests (35 mins)**

*Duration is 25 min*

Click the **START LAB** button (if you have not done so already) and open Cloud Shell as explained in the Setup section. Leave Cloud Shell open for the duration of this workshop.

In this lab, you:

* Clone a git repo that has lab material for this workshop
* Setup the Cloud Shell environment with the tools required for this workshop
* Write a few simple tests to check if you have the prerequisites for running Terraform

Cloning the repo

Get all files associated with this workshop by cloning this repo:

git clone https:*//gitlab.com/binamov/infrastructure-testing.git*

Installing required tools

This workshop uses a few tools and environment variables to make your life easier. These tools are all packaged in a bash script and can be installed by running the following commands:

cd ~/infrastructure-testing

./setup.sh

The following tools are installed:

* terraform
* inspec
* test-kitchen
* kitchen-terraform

Writing your first tests

Google Cloud Shell is already configured with your project credentials. As a warmup, let's write a few tests to make sure that everything is in place and that you can connect to GCP.

Let's start with the local environment. You will use InSpec's os\_envresource to check that you're running inside Cloud Shell and that the GOOGLE\_CLOUD\_PROJECT environment variable is set. Press the Launch Code Editor button at the top right corner of your cloud shell terminal (or use your favorite CLI editor) to edit the file ~/infrastructure-testing/lab\ 1/localtests/controls/prereqs.rb :

title 'prereqs'

describe os\_env('CLOUD\_SHELL') do

its('content') { should eq 'true' }

end

describe os\_env('GOOGLE\_CLOUD\_PROJECT') do

its('content') { should\_not be\_empty }

end

describe command('terraform -version') do

its('exit\_status') { should eq 0 }

its('stdout') { should match /Terraform/ }

end

describe command('kitchen --version') do

its('exit\_status') { should eq 0 }

its('stdout') { should match /Test Kitchen/ }

end

Now run these tests:

cd lab\ 1/

inspec exec localtests

*Output (Do not copy)*

Profile: localtests

Version: 0.1.0

Target: local://

Environment variable CLOUD\_SHELL

✔ content should eq "true"

Environment variable GOOGLE\_CLOUD\_PROJECT

✔ content should not be empty

Command: `terraform -version`

✔ exit\_status should eq 0

✔ stdout should match /Terraform/

Command: `kitchen --version`

✔ exit\_status should eq 0

✔ stdout should match /Test Kitchen/

Test Summary: 6 successful, 0 failures, 0 skipped

You should see something like the above output. At this stage we're expecting the tests to pass. If yours failed (you see Red or something else), retrace your steps back (typos, file locations, exit codes, etc.)

Let's also check that we can connect to GCP. Edit ./gcptests/controls/project.rb and add the following:

title 'GCP project'

gcp\_project = ENV['GOOGLE\_CLOUD\_PROJECT']

describe google\_project(project: gcp\_project) do

it { should exist }

its('project\_id') { should eq gcp\_project }

its('lifecycle\_state') { should eq 'ACTIVE' }

end

Let's run these tests. Note how you provide a -t gcp:// parameter, indicating to InSpec that you're testing against resources in Google Cloud:

inspec exec gcptests -t gcp:*//*

*Output (Do not copy)*

Profile: gcptests

Version: 0.1.0

Target: gcp://default

Project <your project ID>

✔ should exist

✔ project\_id should eq "<your project ID>"

✔ lifecycle\_state should eq "ACTIVE"

Profile: Google Cloud Platform Resource Pack (inspec-gcp)

Version: 0.11.0

Target: gcp://default

No tests executed.

Test Summary: 3 successful, 0 failures, 0 skipped

Everything is green, yay! You should be all set to start using Terraform with GCP.

**END OF LAB 1 - Stop and wait for instructions before proceeding!**

**Lab 2: Write tests for a Terraform GCP deployment**

*Duration is 45 min*

Deploying a Terraform plan

You'll start by deploying a simple Terraform plan. For this lab we are using a [two-tier architecture example](https://github.com/terraform-providers/terraform-provider-google/tree/master/examples/two-tier) that we included with the repo you just cloned. The premise is that you have stateless app servers running behind a load balancer serving traffic.

To get started, change into the ~/infrastructure-testing/lab\ 2and set up terraform:

cd ~/infrastructure-testing/lab\ 2/

terraform init

*Output (Do not copy)*

Initializing provider plugins...

- Checking for available provider plugins on https://releases.hashicorp.com...

- Downloading plugin for provider "google" (2.12.0)...

Terraform has been successfully initialized!

You may now begin working with Terraform. Try running "terraform plan" to see

any changes that are required for your infrastructure. All Terraform commands

should now work.

If you ever set or change modules or backend configuration for Terraform,

rerun this command to reinitialize your working directory. If you forget, other

commands will detect it and remind you to do so if necessary.

Now you're ready to generate a plan:

terraform plan -out plan.out

Take a look at the plan, notice that it will be adding all the GCP resources defined in main.tf as they do not exist yet. If it looks good, go ahead and deploy:

terraform apply plan.out

This process takes about two minutes. It's a good time to stand up and stretch.

When Terraform completes, it will output the instance IPs and the pool public IP. Take a look at how Terraform can test the deployment by running plan again:

terraform plan -out plan.out

*Output (Do not copy)*

No changes. Infrastructure is up-to-date.

This means that Terraform did not detect any differences between your

configuration and real physical resources that exist. As a result, no

actions need to be performed.

See what happens if we run apply again:

terraform apply plan.out

*Output (Do not copy)*

Apply complete! Resources: 0 added, 0 changed, 0 destroyed.

Alright, your app is deployed, congratulations!

Write function tests

Terraform deployed your app and gave you the load balancer IP. You "wrote" the Terraform code, you ran it, you saw there were no errors. Why should we even write function tests?

If you look at the output of the above commands, you can see the resources created by Terraform. One of those is a Firewall. For the purposes of this example, the firewall allows everyone source\_ranges = ["0.0.0.0/0"] to reach the web nodes. This makes sense for a publicly facing app. Similarly, the Firewall allows port 80 inbound.

Chances are that future you, or a colleague, would look at this code a year from now and choose to lock down the source\_ranges or port 80 out of a sense of security. What this would effectively do is break our app - it will not be reachable by everyone.

So let's write a function test to make sure that the right Firewall exists and that it has the right properties. Edit ./function/controls/function.rb and add this control:

gcp\_project = ENV["GOOGLE\_CLOUD\_PROJECT"]

control "default\_firewall" do

describe google\_compute\_firewall(project: gcp\_project, name: 'tf-www-firewall') do

its('allowed\_http?') { should be true }

it { should allow\_ip\_ranges ["0.0.0.0/0"] }

end

end

Let's run these tests against our GCP deployment:

inspec exec function -t gcp:*//*

*Output (Do not copy)*

Profile: function

Version: 0.1.0

Target: gcp://default

✔ default\_firewall: Firewall Rule tf-www-firewall

✔ Firewall Rule tf-www-firewall should allow ip ranges ["0.0.0.0/0"]

✔ Firewall Rule tf-www-firewall allowed\_http? should equal true

Profile: Google Cloud Platform Resource Pack (inspec-gcp)

Version: 0.11.0

Target: gcp://default

No tests executed.

Profile Summary: 1 successful control, 0 control failures, 0 controls skipped

Test Summary: 2 successful, 0 failures, 0 skipped

Congratulations, you just wrote your first Cloud Infrastructure Test!

Alright, your app is deployed, you're probably making money now. Are you though? How do you know for sure?

Running an integration test

Terraform deployed your app and gave you the IP, but how do you know the app is actually up and responsive, actually making you money? You can manually curl the app, but let's write a test instead.

Edit ./integration/controls/integration.rb and add this control:

control "pool\_public\_ip" do

describe http("http://#{attribute('pool\_public\_ip')}/") do

its('status') { should cmp 200 }

its('body') { should match(/Welcome to Resource [012]/) }

end

end

Let's run these tests against our GCP deployment:

inspec exec integration

*Output (Do not copy)*

Profile: integration

Version: 0.1.0

Target: local://

✔ pool\_public\_ip: http GET on http://[POOL PUBLIC IP]/

✔ http GET on http://[POOL PUBLIC IP]/ status should cmp == 200

✔ http GET on http://[POOL PUBLIC IP]/ body should match /Welcome to Resource [012]/

Profile Summary: 1 successful control, 0 control failures, 0 controls skipped

Test Summary: 2 successful, 0 failures, 0 skipped

Everything is green, and green is the color of money!

Introducing infrastructure drift

Let's put on a hat of a disgruntled, unpaid, and uninsured intern that wants to host warez (pirated software) on your infrastructure. They figure that the fastest way to transfer data in and out is using FTP. One opportune moment when you step away from your desk and forget to lock your machine, they walk up and run a command on your computer. So let's create a Firewall Rule that allows Port 21 inbound as the first step to this adversarial takeover:

gcloud compute firewall-rules create takeover --allow tcp:21 --target-tags www-node

*Output (Do not copy)*

Creating firewall...done.

NAME NETWORK DIRECTION PRIORITY ALLOW DENY DISABLED

takeover default INGRESS 1000 tcp:21 False

Writing tests to detect known vulnerabilities

You're back at your desk and suspect interference. You run plan again:

terraform plan -out plan.out

*Output (Do not copy)*

No changes. Infrastructure is up-to-date.

This means that Terraform did not detect any differences between your

configuration and real physical resources that exist. As a result, no

actions need to be performed.

Looks like the Terraform deployment hasn't been tampered with, phew!

The adversary didn't know that you are part of the DevSecOps team and part of your work is writing automation to detect known vulnerabilities, such as a Firewall Rule that allows FTP. Let's edit the file ./gcptests/controls/firewalls.rb:

gcp\_project = ENV["GOOGLE\_CLOUD\_PROJECT"]

control 'gcp-firewalls-no-ftp' do

title 'Ensure no FTP allowed in firewall rules'

google\_compute\_firewalls(project: gcp\_project).firewall\_names.each do |firewall\_name|

describe google\_compute\_firewall(project: gcp\_project, name: firewall\_name) do

it { should\_not allow\_port\_protocol("21", "tcp")}

end

end

end

Let's run these tests against our GCP deployment:

inspec exec gcptests -t gcp:*//*

*Output (Do not copy)*

Profile: gcp-tests

Version: 0.1.0

Target: gcp://default

× gcp-firewalls-no-ftp: Ensure no FTP allowed in firewall rules (2 failed)

✔ Firewall Rule default-allow-icmp should not allow port protocol "21" and "tcp"

× Firewall Rule default-allow-internal should not allow port protocol "21" and "tcp"

expected `Firewall Rule default-allow-internal.allow\_port\_protocol?("21", "tcp")` to return false, got true

✔ Firewall Rule default-allow-rdp should not allow port protocol "21" and "tcp"

✔ Firewall Rule default-allow-ssh should not allow port protocol "21" and "tcp"

× Firewall Rule takeover should not allow port protocol "21" and "tcp"

expected `Firewall Rule takeover.allow\_port\_protocol?("21", "tcp")` to return false, got true

✔ Firewall Rule tf-www-firewall should not allow port protocol "21" and "tcp"

Profile: Google Cloud Platform Resource Pack (inspec-gcp)

Version: 0.11.0

Target: gcp://default

No tests executed.

Profile Summary: 0 successful controls, 1 control failure, 0 controls skipped

Test Summary: 4 successful, 2 failures, 0 skipped

Aha, you detected that two firewall rules break your control for Port 21 being open. You know of one of those rules, default-allow-internal, but then you notice the second rule takeover that should not exist.

You know that you want to keep the default-allow-internal rule, so let's tell inspec not to run this test against that rule. You can do this by using the next keyword. Add the following just before the describeline:

next if firewall\_name == "default-allow-internal"

Your control should now look like this:

gcp\_project = ENV["GOOGLE\_CLOUD\_PROJECT"]

control 'gcp-firewalls-no-ftp' do

title 'Ensure no FTP allowed in firewall rules'

google\_compute\_firewalls(project: gcp\_project).firewall\_names.each do |firewall\_name|

next if firewall\_name == "default-allow-internal"

describe google\_compute\_firewall(project: gcp\_project, name: firewall\_name) do

it { should\_not allow\_port\_protocol("21", "tcp")}

end

end

end

Now run inspec exec gcptests -t gcp:// again:

Profile: gcp-tests

Version: 0.1.0

Target: gcp://default

× gcp-firewalls-no-ftp: Ensure no FTP allowed in firewall rules (1 failed)

✔ Firewall Rule automate-https-firewall should not allow port protocol "21" and "tcp"

✔ Firewall Rule default-allow-icmp should not allow port protocol "21" and "tcp"

✔ Firewall Rule default-allow-rdp should not allow port protocol "21" and "tcp"

✔ Firewall Rule default-allow-ssh should not allow port protocol "21" and "tcp"

× Firewall Rule takeover should not allow port protocol "21" and "tcp"

expected `Firewall Rule takeover.allow\_port\_protocol?("21", "tcp")` to return false, got true

✔ Firewall Rule tf-www-firewall should not allow port protocol "21" and "tcp"

Profile: Google Cloud Platform Resource Pack (inspec-gcp)

Version: 0.11.0

Target: gcp://default

No tests executed.

Profile Summary: 0 successful controls, 1 control failure, 0 controls skipped

Test Summary: 5 successful, 1 failure, 0 skipped

You're down to just one failing test: the one for the takeover rule that was added as part of the attack. Now all you have to do is delete that rule and you're set.

The lesson here is - pay your interns. But also, test your infrastructure for what you know are anti-patterns and vulnerabilities.

Bonus exercises

Try adding some additional controls to your audit profiles. Here are some ideas to get you started:

* verify the configuration of the tf-www-basic-check health check (hint: use the [google\_compute\_http\_health\_check](https://github.com/inspec/inspec-gcp/blob/master/docs/resources/google_compute_http_health_check.md" \t "_blank) resource)
* verify that the tf-www-target-pool exists and contains all three instances (hint: [google\_compute\_target\_pool](https://github.com/inspec/inspec-gcp/blob/master/docs/resources/google_compute_target_pool.md" \t "_blank))
* verify that you have the expected number of VMs (hint: try the [google\_compute\_instances](https://github.com/inspec/inspec-gcp/blob/master/docs/resources/google_compute_instances.md" \t "_blank) resource. Unless you changed the Terraform code, your instances are in the us-central1-f zone)
* Take a look at [the InSpec-GCP examples](https://github.com/inspec/inspec-gcp/tree/master/test/integration/verify/controls) for more inspiration.

**END OF LAB 2 - Stop and wait for instructions before proceeding!**

**Lab 3: Streamline the deploy - test - change - test - destroy process for a GKE deployment with kitchen-terraform**

In the previous two labs you saw that writing infrastructure tests along your infrastructure code can help you test the various resources for their state, the entire deployment for function, and even for known vulnerabilities.

The drawback of introducing testing, at least at the beginning, is it adds more steps to a workflow. Here's what it may look like in a project:

* set up terraform with terraform init
* provision your infrastructure with terraform apply
* run tests with inspec exec gcptests -t gcp://
* see failing tests; fix the terraform code
* run terraform apply again
* run inspec again
* see that the tests pass
* clean up with terraform destroy

This cycle is tedious and complicated and it's easy to get lost in the context switching.

Enter kitchen-terraform. It's based on the popular test-kitchenproject and is designed to make the above kind of testing workflow easy. Let's give it a try.

Start by cleaning up from the previous lab:

terraform destroy -auto-approve

This will take a minute or two. When it's done, switch into ~/infrastructure-testing/lab\ 3 and take a look at the test-kitchen configuration:

cd ~/infrastructure-testing/lab\ 3/

kitchen list

*Output (Do not copy)*

Instance Driver Provisioner Verifier Transport Last Action Last Error

verify-k8s Terraform Terraform Terraform Ssh <Not Created> <None>

We can see that we have a verify-k8s instance using the Terraformdriver and SSH as the transport. This is a good way to see if the kitchen-terraform configuration file has any syntax errors.

From here we can get straight to Provisioning/Converging our Terraform:

kitchen converge

*Output (Do not copy)*

Apply complete! Resources: 1 added, 0 changed, 0 destroyed.

Finished converging <verify-k8s> (3m6.17s).

-----> Kitchen is finished. (3m11.67s)

This is similar to terraform apply and just like before you'll see that it's actually running terraform for you. You use kitchen converge to apply any Terraform code changes you've made. You can see that we didn't have to run a terraform init and in fact you will see that test-kitchen runs terraform init during setup.

Once kitchen is done converging your infrastructure, you can run an InSpec test against your deployment with:

kitchen verify

*Output (Do not copy)*

system: Verifying

Profile: gcp-kubetesting

Version: (not specified)

Target: gcp://default

✔ gcp-container-cluster-node-pools-1.0: Ensure container cluster node pools have the correct properties in bulk.

✔ google\_container\_clusters should exist

✔ google\_container\_clusters count should be <= 10

✔ google\_container\_clusters cluster\_names should include "gcp-inspec-kube-cluster"

Profile: Google Cloud Platform Resource Pack (inspec-gcp)

Version: 0.11.0

Target: gcp://default

No tests executed.

Profile Summary: 1 successful control, 0 control failures, 0 controls skipped

Test Summary: 3 successful, 0 failures, 0 skipped

Finished verifying <verify-k8s> (0m16.25s).

-----> Kitchen is finished. (0m19.49s)

Now that you understand the basic mechanics of kitchen-terraform, let's write another test.

Open up the file ./test/integration/verify/controls/k8s.rb in your text editor and add this control to the bottom of the file:

control 'gcp-container-cluster' do

describe google\_container\_cluster(project: gcp\_project\_id, zone: gcp\_kube\_cluster\_zone, name: gcp\_kube\_cluster\_name) do

it { should exist }

its('master\_auth.username'){ should eq "gcp-inspec-kube-admin"}

end

end

Save the file, and run kitchen verify again:

*Output (Do not copy)*

system: Verifying

Profile: gcp-kubetesting

Version: (not specified)

Target: gcp://default

✔ gcp-container-cluster-node-pools-1.0: Ensure container cluster node pools have the correct properties in bulk.

✔ google\_container\_clusters should exist

✔ google\_container\_clusters count should be <= 10

✔ google\_container\_clusters cluster\_names should include "gcp-inspec-kube-cluster"

✔ gcp-container-cluster: Cluster gcp-inspec-kube-cluster

✔ Cluster gcp-inspec-kube-cluster should exist

✔ Cluster gcp-inspec-kube-cluster master\_auth.username should eq "gcp-inspec-kube-admin"

Profile: Google Cloud Platform Resource Pack (inspec-gcp)

Version: 0.11.0

Target: gcp://default

No tests executed.

Profile Summary: 2 successful controls, 0 control failures, 0 controls skipped

Test Summary: 5 successful, 0 failures, 0 skipped

Finished verifying <verify-k8s> (0m16.66s).

-----> Kitchen is finished. (0m20.05s)

Easy, right?

Let's go through one more example. Suppose that our Trust and Compliance team issued a requirement that **All our workloads should live across the four availability zones of the us-central1 region and nowhere else**. Let's express that as a control, by adding the following code to the file ./test/integration/verify/controls/k8s.rb :

control 'cluster-data-locality' do

impact 1.0

title 'Ensure cluster is only in us-central1 region'

describe google\_container\_cluster(project: gcp\_project\_id, zone: gcp\_kube\_cluster\_zone, name: gcp\_kube\_cluster\_name) do

its('locations.sort'){ should cmp ["us-central1-a", "us-central1-b", "us-central1-c", "us-central1-f"].sort }

end

end

Save the file, and run kitchen verify again:

*Output (Do not copy)*

system: Verifying

Profile: gcp-kubetesting

Version: (not specified)

Target: gcp://default

✔ gcp-container-cluster-node-pools-1.0: Ensure container cluster node pools have the correct properties in bulk.

✔ google\_container\_clusters should exist

✔ google\_container\_clusters count should be <= 10

✔ google\_container\_clusters cluster\_names should include "gcp-inspec-kube-cluster"

✔ gcp-container-cluster: Cluster gcp-inspec-kube-cluster

✔ Cluster gcp-inspec-kube-cluster should exist

✔ Cluster gcp-inspec-kube-cluster master\_auth.username should eq "gcp-inspec-kube-admin"

× cluster-data-locality: Ensure cluster is only in us-central1 region

× Cluster gcp-inspec-kube-cluster locations.sort should cmp == ["us-central1-a", "us-central1-b", "us-central1-c", "us-central1-f"]

expected: ["us-central1-a", "us-central1-b", "us-central1-c", "us-central1-f"]

got: ["us-central1-a", "us-central1-b", "us-central1-c"]

(compared using `cmp` matcher)

Profile: Google Cloud Platform Resource Pack (inspec-gcp)

Version: 0.11.0

Target: gcp://default

No tests executed.

Profile Summary: 2 successful controls, 1 control failure, 0 controls skipped

Test Summary: 5 successful, 1 failure, 0 skipped

Looks like our workload did not pass our control. When we look closer, we can see that our control was expecting all four zones - a, b, c and f, while our code specifies just a, b and c. We can modify our control, but that would constrain the zones in which we can run our workload, it also wouldn't satisfy the expectations of our Trust and Compliance team. So instead, let's correct our Terraform code by adding the missing zone. Open the file main.tf and add the missing zone to the node\_locations block on lines 22-25, so it looks like this::

node\_locations = [

"us-central1-a",

"us-central1-c",

"us-central1-f",

]

Save the file. We just changed our Terraform code. With kitchen, just run kitchen converge again to modify our running cluster:

*Output (Do not copy)*

node\_locations.#: "2" => "3"

node\_locations.1655143457: "" => "us-central1-f"

node\_locations.533760100: "us-central1-c" => "us-central1-c"

node\_locations.770108134: "us-central1-a" => "us-central1-a"

google\_container\_cluster.primary: Modifications complete after 1m9s (ID: gcp-inspec-kube-cluster)

Apply complete! Resources: 0 added, 1 changed, 0 destroyed.

Finished converging <verify-k8s> (1m12.38s).

-----> Kitchen is finished. (1m15.59s)

After converge is done, go ahead and run kitchen converge again - you should see that the deployment is in our desired state and no additional changes are required. Once the converge is complete, let's re-run our tests with a kitchen verify again:

*Output (Do not copy)*

system: Verifying

Profile: gcp-kubetesting

Version: (not specified)

Target: gcp://default

✔ gcp-container-cluster-node-pools-1.0: Ensure container cluster node pools have the correct properties in bulk.

✔ google\_container\_clusters should exist

✔ google\_container\_clusters count should be <= 10

✔ google\_container\_clusters cluster\_names should include "gcp-inspec-kube-cluster"

✔ gcp-container-cluster: Cluster gcp-inspec-kube-cluster

✔ Cluster gcp-inspec-kube-cluster should exist

✔ Cluster gcp-inspec-kube-cluster master\_auth.username should eq "gcp-inspec-kube-admin"

✔ cluster-data-locality: Ensure cluster is only in us-central1 region

✔ Cluster gcp-inspec-kube-cluster locations.sort should cmp == ["us-central1-a", "us-central1-b", "us-central1-c", "us-central1-f"]

Profile: Google Cloud Platform Resource Pack (inspec-gcp)

Version: 0.11.0

Target: gcp://default

No tests executed.

Profile Summary: 3 successful controls, 0 control failures, 0 controls skipped

Test Summary: 6 successful, 0 failures, 0 skipped

Finished verifying <verify-k8s> (0m16.58s).

-----> Kitchen is finished. (0m19.80s)

All tests are passing. We now have a Kubernetes deployment on GKE that is functional AND compliant with the requirements of our Trust & Compliance team!

Exercises

**Exercise 1:**Experiment with introducing drift and catching it with kitchen:

* Edit the gcp-container-cluster-node-pools-1.0 control to assert a specific number of clusters
* Modify main.tf to add additional clusters and re-run kitchen verify again
* Edit the control again to get it to pass
* Hint: { should be\_all "RUNNING" } is one way to make sure that all of your clusters are running

**Exercise 2:**Add additional tests to gcp-container-cluster to assert tighter controls over how the cluster is built. See [the docs](https://github.com/inspec/inspec-gcp/blob/master/docs/resources/google_container_cluster.md) for ideas.

* If you did the first bonus exercise, add additional controls for your new clusters. Hint: you can create these manually or you can iterate over the clusters just like we did in lab 2. Each approach has tradeoffs; try both!

**Want more?**Take a look at [the InSpec-GCP examples](https://github.com/inspec/inspec-gcp/tree/master/test/integration/verify/controls) and write another several tests for this Terraform deployment. Use kitchen verify to run the tests.

Cleaning up

Finally, once you have both your Terraform and InSpec code in the shape that you want them to be, you can simply destroy the entire deployment:

kitchen destroy

The beautiful thing about the kitchen commands is that you can run converge from a freshly cloned repo, without having to do init nor plan – those will be automagically included, so that you fast-forward to the valuable stage of deploying and verifying.

Now that you understand the mechanics of kitchen-terraform , see how easy it is to run the entire init → apply → test → destroy loop with a single command (eg from your cloud shell, developer workstation, or automatically by your Continuous Integration worker):

kitchen test

Runs about 5 minutes end to end.

**Congratulations!**