Build Infrastructure

Terraform comes pre-installed in Cloud Shell. With Terraform already installed, you can dive right in and create some infrastructure.

Start by creating your example configuration to a file named main.tf. Terraform recognizes files ending in .tf or .tf.json as configuration files and will load them when it runs.

1.Create the main.tf file:

```
touch main.tf
```

content_copy

2.Click the **Open Editor** button on the toolbar of Cloud Shell. (You can switch between Cloud Shell and the code editor by using the **Open Editor** and **Open Terminal** icons as required, or click the **Open in new window** button to leave the Editor open in a separate tab).

3.In the Editor, add the following content to the main.tf file. Make sure to replace <PROJECT_ID> with the lab's Project ID:

```
terraform {
  required_providers {
    google = {
      source = "hashicorp/google"
    }
  }
}
provider "google" {
  version = "3.5.0"
  project = "<PROJECT_ID>"
  region  = "us-central1"
  zone    = "us-central1-c"
}
resource "google_compute_network" "vpc_network" {
  name = "terraform-network"
}
```

content_copy

Tip: To use this snippet with Terraform 0.12, remove the terraform {} block.

Terraform Block

The terraform {} block is required so Terraform knows which provider to download from the Terraform Registry. In the configuration above, the google provider's source is defined as hashicorp/google which is shorthand for registry.terraform.io/hashicorp/google.

You can also assign a version to each provider defined in the required_providers block. The version argument is optional, but recommended. It is used to constrain the provider to a specific version or a range of versions in order to prevent downloading a new provider that may possibly contain breaking changes. If the version isn't specified, Terraform will automatically download the most recent provider during initialization.

To learn more, reference the provider source documentation.

Providers

The provider block is used to configure the named provider, in this case google. A provider is responsible for creating and managing resources. Multiple provider blocks can exist if a Terraform configuration manages resources from different providers.

Initialization

The first command to run for a new configuration -- or after checking out an existing configuration from version control -- is terraform init, which initializes various local settings and data that will be used by subsequent commands.

1.Initialize your new Terraform configuration by running the terraform init command in the same directory as your main.tf file:

```
terraform init
```

content_copy

Creating Resources

1.Apply you configuration now by running the command terraform apply:

```
terraform apply
```

content_copy

The output has a + next to resource "google_compute_network" "vpc_network", meaning that Terraform will create this resource. Beneath that, it shows the attributes that will be set. When the value displayed is (known after apply), it means that the value won't be known until the resource is created.

If the plan was created successfully, Terraform will now pause and wait for approval before proceeding. If anything in the plan seems incorrect or dangerous, it is safe to abort here with no changes made to your infrastructure.

If terraform apply failed with an error, read the error message and fix the error that occurred.

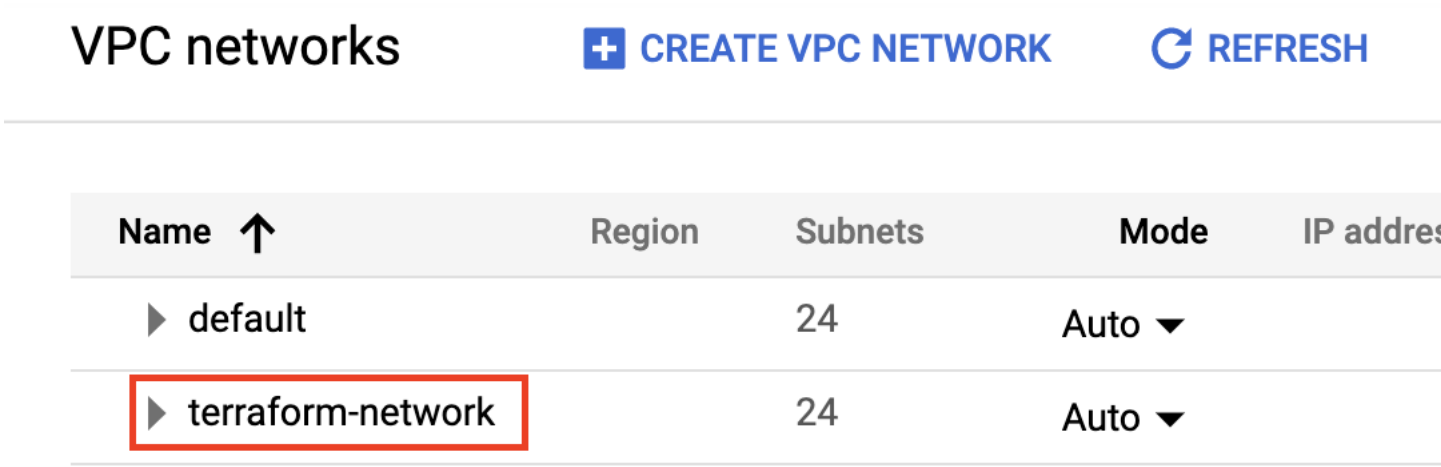
2.The plan looks acceptable here, so type yes at the confirmation prompt to proceed.
Executing the plan will take a few minutes since Terraform waits for the network to be created successfully:

```
# ...
  Enter a value: yes
google_compute_network.vpc_network: Creating...
google_compute_network.vpc_network: Still creating... [10s elapsed]
google_compute_network.vpc_network: Still creating... [20s elapsed]
google_compute_network.vpc_network: Still creating... [30s elapsed]
google_compute_network.vpc_network: Still creating... [40s elapsed]
google_compute_network.vpc_network: Still creating... [50s elapsed]
google_compute_network.vpc_network: Creation complete after 58s [id=terraform-network]
Apply complete! Resources: 1 added, 0 changed, 0 destroyed.
```

content_copy

After this, Terraform is all done! You can go to the Cloud Console to see the network you have provisioned.

3.In the Console, from the **Navigation menu**, navigate to **VPC network**. You will see the terraform-network has been provisioned.



4.In Cloud Shell run the terraform show command to inspect the current state.

```
terraform show
```

content_copy

These values can be referenced to configure other resources or outputs, which will be covered later in this lab.

Click Check my progress to verify the objective.

Change Infrastructure

In the previous section, you created basic infrastructure with Terraform: a VPC network. In this section, you're going to modify your configuration, and see how Terraform handles change.

Infrastructure is continuously evolving, and Terraform was built to help manage and enact that change. As you change Terraform configurations, Terraform

builds an execution plan that only modifies what is necessary to reach your desired state.

By using Terraform to change infrastructure, you can version control not only your configurations but also your state so you can see how the infrastructure evolves over time.

Adding Resources

You can add new resources by adding them to your Terraform configuration and running terraform apply to provision them.

1.In the Editor, add a compute instance resource to main.tf:

```
resource "google_compute_instance" "vm_instance" {
  name         = "terraform-instance"
  machine_type = "f1-micro"
  boot_disk {
    initialize_params {
      image = "debian-cloud/debian-9"
    }
  }
  network_interface {
    network = google_compute_network.vpc_network.name
    access_config {
    }
  }
}
content_copy
```

This resource includes a few more arguments. The name and machine type are simple strings, but boot_disk and network_interface are more complex blocks. You can see all of the available options in the documentation.

For this example, your compute instance will use a Debian operating system, and will be connected to the VPC Network you created earlier. Notice how this configuration refers to the network's name property with google_compute_network.vpc_network.name -- google_compute_network.vpc_n

etwork is the ID, matching the values in the block that defines the network, and name is a property of that resource.

The presence of the access_config block, even without any arguments, ensures that the instance will be accessible over the internet.

2.Now run terraform apply to create the compute instance:

```
terraform apply
```

content_copy

Once again, answer yes to the confirmation prompt.

This is a fairly straightforward change - you added a "google_compute_instance" resource named "vm_instance" to your configuration, and Terraform created the resource in Google Cloud.

Changing Resources

In addition to creating resources, Terraform can also make changes to those resources.

1.Add a tags argument to your "vm_instance" so that it looks like this:

```
resource "google_compute_instance" "vm_instance" {
  name         = "terraform-instance"
  machine_type = "f1-micro"
  tags         = ["web", "dev"]
  # ...
}
```

content_copy

2.Run terraform apply again to update the instance.

```
terraform apply
```

content_copy

3.The prefix ~ means that Terraform will update the resource in-place. You can go and apply this change now by responding yes, and Terraform will add the tags to your instance.
Click Check my progress to verify the objective.

Destructive Changes

A destructive change is a change that requires the provider to replace the existing resource rather than updating it. This usually happens because the cloud provider doesn't support updating the resource in the way described by your configuration.

Changing the disk image of your instance is one example of a destructive change.

1.Edit the boot_disk block inside the vm_instance resource in your configuration file and change it to the following:

```
  boot_disk {
    initialize_params {
      image = "cos-cloud/cos-stable"
    }
  }
content_copy
```

2.Now run terraform apply again to see how Terraform will apply this change to the existing resources:

```
terraform apply
content_copy
```

The prefix -/+ means that Terraform will destroy and recreate the resource, rather than updating it in-place. While some attributes can be updated in-place (which are shown with the ~ prefix), changing the boot disk image for an instance requires recreating it. Terraform and the Google Cloud provider handle these details for you, and the execution plan makes it clear what Terraform will do.

Additionally, the execution plan shows that the disk image change is what required your instance to be replaced. Using this information, you can adjust your changes to possibly avoid destroy/create updates if they are not acceptable in some situations.

3.Once again, Terraform prompts for approval of the execution plan before proceeding. Answer yes to execute the planned steps.
As indicated by the execution plan, Terraform first destroyed the existing instance and then created a new one in its place. You can use terraform show again to see the new values associated with this instance.

Destroy Infrastructure

You have now seen how to build and change infrastructure. Before moving on to creating multiple resources and showing resource dependencies, you will see how to completely destroy your Terraform-managed infrastructure.

Destroying your infrastructure is a rare event in production environments. But if you're using Terraform to spin up multiple environments such as development, testing, and staging, then destroying is often a useful action.

Resources can be destroyed using the `terraform destroy` command, which is similar to `terraform apply` but it behaves as if all of the resources have been removed from the configuration.

1.Try the `terraform destroy` command. Answer `yes` to execute this plan and destroy the infrastructure:

```
terraform destroy
```

content_copy

The - prefix indicates that the instance and the network will be destroyed. As with apply, Terraform shows its execution plan and waits for approval before making any changes.

Just like with `terraform apply`, Terraform determines the order in which things must be destroyed. Google Cloud won't allow a VPC network to be deleted if there are resources still in it, so Terraform waits until the instance is destroyed before destroying the network. When performing operations, Terraform creates a dependency graph to determine the correct order of operations. In more complicated cases with multiple resources, Terraform will perform operations in parallel when it's safe to do so.

Click Check my progress to verify the objective.

Create Resource Dependencies

In this section, you will learn more about resource dependencies and how to use resource parameters to share information about one resource with other resources.

Real-world infrastructure has a diverse set of resources and resource types. Terraform configurations can contain multiple resources, multiple resource types, and these types can even span multiple providers.

In this section, you will be shown a basic example of how to configure multiple resources and how to use resource attributes to configure other resources.

1.Recreate your network and instance. After you respond to the prompt with yes, the resources will be created.

```
terraform apply
```

content_copy

Assigning a Static IP Address

1.Now add to your configuration by assigning a static IP to the VM instance in main.tf.

```
resource "google_compute_address" "vm_static_ip" {
  name = "terraform-static-ip"
}
```

content_copy

This should look familiar from the earlier example of adding a VM instance resource, except this time you're creating a "google_compute_address" resource type. This resource type allocates a reserved IP address to your project.

2.Next, run terraform plan.

```
terraform plan
```

content_copy

You can see what will be created with terraform plan:

```
$ terraform plan
Refreshing Terraform state in-memory prior to plan...
The refreshed state will be used to calculate this plan, but will not be
persisted to local or remote state storage.
google_compute_network.vpc_network: Refreshing state... [id=terraform-network]
google_compute_instance.vm_instance: Refreshing state... [id=terraform-instance]
------------------------------------------------------------------------
An execution plan has been generated and is shown below.
Resource actions are indicated with the following symbols:
  + create
Terraform will perform the following actions:
  # google_compute_address.vm_static_ip will be created
  + resource "google_compute_address" "vm_static_ip" {
      + address           = (known after apply)
      + address_type      = "EXTERNAL"
      + creation_timestamp = (known after apply)
      + id                = (known after apply)
      + name              = "terraform-static-ip"
      + network_tier      = (known after apply)
      + project           = (known after apply)
      + region            = (known after apply)
      + self_link         = (known after apply)
      + subnetwork        = (known after apply)
      + users             = (known after apply)
    }
Plan: 1 to add, 0 to change, 0 to destroy.
------------------------------------------------------------------------
Note: You didn't specify an "-out" parameter to save this plan, so Terraform can't guarantee that
exactly these actions will be performed if
"terraform apply" is subsequently run.
```

content_copy

Unlike terraform apply, the plan command will only show what would be changed, and never actually apply the changes directly. Notice that the only change you have made so far is to add a static IP. Next, you need to attach the IP address to your instance.

3.Update the network_interface configuration for your instance like so:

```
  network_interface {
    network = google_compute_network.vpc_network.self_link
    access_config {
      nat_ip = google_compute_address.vm_static_ip.address
    }
  }
content_copy
```

The access_config block has several optional arguments, and in this case you'll set nat_ip to be the static IP address. When Terraform reads this configuration, it will:

•Ensure that vm_static_ip is created before vm_instance

•Save the properties of vm_static_ip in the state

•Set nat_ip to the value of the vm_static_ip.address property

4.Run terraform plan again, but this time, save the plan:

```
terraform plan -out static_ip
content_copy
```

Saving the plan this way ensures that you can apply exactly the same plan in the future. If you try to apply the file created by the plan, Terraform will first check to make sure the exact same set of changes will be made before applying the plan.

In this case, you can see that Terraform will create a new google_compute_address and update the existing VM to use it.

5.Run terraform apply "static_ip" to see how Terraform plans to apply this change:

```
terraform apply "static_ip"
content_copy
```

As shown above, Terraform created the static IP before modifying the VM instance. Due to the interpolation expression that passes the IP address to the instance's network interface configuration, Terraform is able to infer a dependency, and knows it must create the static IP before updating the instance.

Click Check my progress to verify the objective.

Implicit and Explicit Dependencies

By studying the resource attributes used in interpolation expressions, Terraform can automatically infer when one resource depends on another. In the example above, the reference to google_compute_address.vm_static_ip.address creates an implicit dependency on the google_compute_address named vm_static_ip.

Terraform uses this dependency information to determine the correct order in which to create and update different resources. In the example above, Terraform knows that the vm_static_ip must be created before the vm_instance is updated to use it.

Implicit dependencies via interpolation expressions are the primary way to inform Terraform about these relationships, and should be used whenever possible.

Sometimes there are dependencies between resources that are not visible to Terraform. The depends_on argument can be added to any resource and accepts a list of resources to create explicit dependencies for.

For example, perhaps an application you will run on your instance expects to use a specific Cloud Storage bucket, but that dependency is configured inside the application code and thus not visible to Terraform. In that case, you can use depends_on to explicitly declare the dependency.

1.Add a Cloud Storage bucket and an instance with an explicit dependency on the bucket by adding the following to main.tf:

```
# New resource for the storage bucket our application will use.
resource "google_storage_bucket" "example_bucket" {
  name     = "<UNIQUE-BUCKET-NAME>"
  location = "US"
  website {
    main_page_suffix = "index.html"
    not_found_page   = "404.html"
  }
}
# Create a new instance that uses the bucket
resource "google_compute_instance" "another_instance" {
  # Tells Terraform that this VM instance must be created only after the
  # storage bucket has been created.
  depends_on = [google_storage_bucket.example_bucket]
```

```
  name         = "terraform-instance-2"
  machine_type = "f1-micro"
  boot_disk {
    initialize_params {
      image = "cos-cloud/cos-stable"
    }
  }
  network_interface {
    network = google_compute_network.vpc_network.self_link
    access_config {
    }
  }
}
content_copy
```

UNIQUE-BUCKET-NAME with a unique, valid name for a bucket. Using your name and the date is usually a good way to guess a unique bucket name.

You may wonder where in your configuration these resources should go. The order that resources are defined in a terraform configuration file has no effect on how Terraform applies your changes. Organize your configuration files in a way that makes the most sense for you and your team.

2.Now run terraform plan and terraform apply to see these changes in action:

```
terraform plan
terraform apply
content_copy
```

3.Before moving on, remove these new resources from your configuration and run terraform apply once again to destroy them. You won't use the bucket or the second instance any further in the getting started guide.

Provision Infrastructure

The compute instance you launched at this point is based on the Google image given, but has no additional software installed or configuration applied.

Google Cloud allows customers to manage their own custom operating system images. This can be a great way to ensure the instances you provision with Terraform are pre-configured based on your needs. Packer is the perfect tool for this and includes a builder for Google Cloud.

Terraform uses provisioners to upload files, run shell scripts, or install and trigger other software like configuration management tools.

Defining a Provisioner

1.To define a provisioner, modify the resource block defining the first vm_instance in your configuration to look like the following:

```
resource "google_compute_instance" "vm_instance" {
   name        = "terraform-instance"
   machine_type = "f1-micro"
   tags        = ["web", "dev"]
   provisioner "local-exec" {
      command = "echo ${google_compute_instance.vm_instance.name}:  $
{google_compute_instance.vm_instance.network_interface[0].access_config[0].nat_ip} >>
ip_address.txt"
   }
   # ...
}
content_copy
```

This adds a provisioner block within the resource block.
Multiple provisioner blocks can be added to define multiple provisioning steps.
Terraform supports many provisioners, but for this example you are using the local-exec provisioner.

The local-exec provisioner executes a command locally on the machine running Terraform, not the VM instance itself. You're using this provisioner versus the others so we don't have to worry about specifying any connection info right now.

This also shows a more complex example of string interpolation than you've seen before. Each VM instance can have multiple network interfaces, so refer to the first one with network_interface[0], count starting from 0, as most programming languages do. Each network interface can have multiple access_config blocks as well, so once again you specify the first one.

2.Run terraform apply. At this point, the output may be confusing at first.

```
terraform apply
```

content_copy

Terraform found nothing to do - and if you check, you'll find that there's no ip_address.txt file on your local machine.

Terraform treats provisioners differently from other arguments. Provisioners only run when a resource is created, but adding a provisioner does not force that resource to be destroyed and recreated.

3.Use terraform taint to tell Terraform to recreate the instance:

```
terraform taint google_compute_instance.vm_instance
```

content_copy

A tainted resource will be destroyed and recreated during the next apply.

4.Run terraform apply now:

```
terraform apply
```

content_copy

5.Verify everything worked by looking at the contents of the ip_address.txt file. It contains the IP address, just as you asked.

Failed Provisioners and Tainted Resources

If a resource is successfully created but fails a provisioning step, Terraform will error and mark the resource as tainted. A resource that is tainted still exists, but shouldn't be considered safe to use, since provisioning failed.

When you generate your next execution plan, Terraform will remove any tainted resources and create new resources, attempting to provision them again after creation.

Destroy Provisioners

Provisioners can also be defined that run only during a destroy operation. These are useful for performing system cleanup, extracting data, etc.

For many resources, using built-in cleanup mechanisms is recommended if possible (such as init scripts), but provisioners can be used if necessary.

This lab won't show any destroy provisioner examples. If you need to use destroy provisioners, please see the provisioner documentation.

Congratulations!

In this lab, you learned how to build, change, and destroy infrastructure with Terraform. You then created resource dependencies, and provisioned basic infrastructure with Terraform configuration files.

Finish Your Quest

This self-paced lab is part of the Qwiklabs Automating Infrastructure on Google Cloud with Terraform quest. A quest is a series of related labs that form a learning path. Completing a quest earns you a badge to recognize your achievement. You can make your badge (or badges) public and link to them in your online resume or social media account. Enroll in a quest and get immediate completion credit if you've taken this lab. See other available Qwiklabs Quests.

Take your next lab

Continue your quest with the next lab, Interact with Terraform Modules.

Next steps/Learn More

Be sure to check out the following to receive more hands-on practice with Terraform:

•Hashicorp on the Google Cloud Marketplace!
•Hashicorp Learn
•Terraform Community
•Terraform Google Examples

Google Cloud Training & Certification

...helps you make the most of Google Cloud technologies. Our classes include technical skills and best practices to help you get up to speed quickly and continue your learning journey. We offer fundamental to advanced level training, with on-demand, live, and virtual options to suit your busy schedule. Certifications help you validate and prove your skill and expertise in Google Cloud technologies.

Manual Last Updated August 28, 2021
Lab Last Tested Feb 23, 2021