Lab Guide

Build Cloud Infrastructure with Terraform

Version 1.0

Christopher Parent, Instructor

Table of Contents

Lab Guide Overview.. 3
Lab #1: Getting Started with Terraform....................................... 5
Lab #2: Terraform Variables and Interpolations 10
Lab #3: Resource Dependencies 22
Lab #4: Terraform Provisioners 26
Lab #5: Data Sources 34
Lab #6: Terraform Modules 36
Lab #7: Terraform State.. 40
Extra - Using an AWS Credentials File with Terraform................. 45
References.. 47

Lab Guide Overview

This lab guide will teach you how easy and efficient it us to use Hashicorp Terraform to create and manage infrastructure. Each lab builds upon the previous, so it is highly recommended you perform all the labs in sequence.

The References section of this document contains links to the software needed for this lab, as well as links to online documentation for reference.

Organization

Each lab is organized in the following structure:

- **Skills Learned** describes what you will get out of the lab
- **Overview** describes the overall details of the lab
- **Instructions** provide the details steps needed to perform the lab

Formatting Convention

The lab uses various text formatting to convey meaning and action:

- **Bold** font will be used to emphasize actions to be taken:
    - **Click** the button
    - **Open** a terminal window
- `Courier font will be used to emphasize command syntax and command line execution`

General System Requirements

Terraform is a fairly lightweight single binary used as a client to talk to cloud service providers, therefore system requirements are fairly minimal.

For this lab guide, the following is required:

- Internet connection
- A cloud account with AWS and any other cloud providers mentioned in this lab.
- Modern OS such as Linux, Windows, or OSX
- Text editor (Atom, vi, Notepad)

Operating System Requirements

Terraform has native support for a variety of operating systems so feel free to choose the one that is appropriate for you.

The exercises in this lab guide have been written using Linux as the client operating system, however the Terraform command line syntax and Terraform HCL used to model resources in the cloud is platform agnostic.

The author will make every attempt to show both Windows and *Nix-specific commands where possible, such as setting of environment variables.

Links to required software can be found in the References section of this lab guide.

DISCLAIMER – The author of this document nor any of its affiliates or partners are not responsible for any costs incurred during the execution of this lab guide or any of its related material.

Need Help?

If you need help with the labs or have questions please contact me directly at learn@chrisparent.io.

Lab #1: Getting Started with Terraform

Duration 30 minutes

Overview

In this lab you will get familiar with Terraform by creating a compute instance in AWS.

If you already have an AWS account, then there is no need to create another account. You may follow the steps to create an additional API user with keys for use with this lab.

Skills Learned

At the end of this exercise you will have prepared your workstation to run Docker.

- Create an account in AWS (Free Tier)
- Download and install Terraform
- Create a Terraform configuration
- Create and destroy an EC2 instance in AWS

Instructions

1	**Create an AWS Free Tier Account**
1.1	Sign up for a Free Tier account here: https://aws.alearncdmazon.com/free/
2	**Create an AWS API Key**
2.1	To use the Terraform AWS provider you need to have created a set of API keys for accessing AWS. Once you have your AWS account, log in to the AWS management console.
2.2	Navigate to Services > IAM > Users
2.3	Click 'Add user' to create a new user.
2.4	Specify a user name.
2.5	Under Select AWS access type, check the box for Programmatic Access
2.6	Click 'Next:Permissions'
2.7	Click 'Attach existing policies directly' at the top of the page.
2.8	Select the 'AdministratorAccess' policy. This policy grants full administrators rights to this account.
2.9	Click 'Next: Review'
2.10	If everything looks right, click 'Create user.'

2.11	AWS will generate an Access key ID and a Secret access key. Save both of these values for later. You will need to configure Terraform to use both of these values to authenticate against AWS.
3	**Download and Install Terraform**
3.1	You can download Terraform for your system from here: https://www.terraform.io/downloads.html Terraform is packaged as a single binary.
3.2	Unzip the downloaded archive to a location of your choosing.

3.3	Add the terraform binary to your system's Path environment variable. Be sure to replace /path/to/terraform with the actual path from the previous step.
	
	For bash shells
	
	`$ export PATH=%PATH%:/path/to/terraform`
	
	For Windows command prompt
	
	`C:\> setx path "%path%;c:\path\to\terraform\"`

3.4	Verify terraform is installed by opening a new terminal window or command prompt and run 'terraform'. `$ terraform`
4	**Create a Development workspace**
4.1	In a terminal window or command prompt, create a new directory. This new directory is where you will create your Terraform configurations. `$ mkdir -p learntf/lab1`
4.2	Using for favorite text editor, create a new text file in the lab1 directory named lab1.tf.
4.3	Add the following code to lab1.tf. Be sure to substitute actual values for below. The region specifies in which data center resources will be created. *This lab guide will use us-region-1. OS images are specific to a particular region. If you must use a different region, then you must also find OS images that are available in the new region.* *You can get a full list of regions and their region codes here:* https://docs.aws.amazon.com/general/latest/gr/rande.html

```
provider "aws" {
  access_key = "<INSERT YOUR AWS ACCESS KEY>"
  secret_key = "<INSERT YOUR AWS SECRET KEY>"
  region     = “us-east-1”
}
```

The code block above describes which cloud provider you want to use, the authentication credentials, and the region in which you want to create resources.

DISCLAIMER. Hard coding AWS credentials in source files is an extremely bad practice. Do not place this file under source control or store it anywhere where anyone else may get access to it. You will learn shortly how to use environment variables to set your AWS credentials.

4.4 Save the file.

4.5 In the terminal window initialize the development workspace by running the following command in the lab1 directory

```
[user@host lab1]$ terraform init
```

This command will download the AWS terraform provider and prepare the directory for use with Terraform.

5 Create an AWS EC2 Instance using Terraform

5.1 In the following steps you will create a Terraform configuration which is one or more text files written in HCL for managing compute and other infrastructure resources.

Open lab1.tf in your text editor and add the following snippet of code after the provider block. Save the file after making the change.

```
resource "aws_instance" "myexample" {
  ami           = "ami-b374d5a5"
  instance_type = "t2.micro"
}
```

This configuration will create a resource of type ‘aws_instance’ named ‘myexample.’ The name is internal to Terraform and will be used later on as a reference to the compute instance.

The attributes describe the EC2 instance that we want to create. Specifically, ami specifies the OS image to load and the instance_type specifies the shape or size of the compute resource.

	These attributes and their values are specific to AWS. Other providers may use similar but different attributes and values.
5.2	Next let's create a build execution plan by running terraform plan. `$ terraform plan` This command determines what steps terraform will take to reach the desired state of the resources defined in the configuration.
5.3	With the build execution plan created, we can now create our EC2 instance. `$ terraform apply` Terraform will now create the EC2 instance in AWS. Terraform will block until the instance is created.
5.4	You can verify the instance is created by logging into the AWS web console and navigating to Services > EC2 > Instances.
6	**Change an AWS EC2 Instance**
	Save the file.
6.1	Back in the terminal re-run terraform plan to generate the build execution plan. Observe the output from the plan and take note of the steps terraform will take. You should see terraform will first destroy then re-create the instance, noted by the -/+ characters next to the name of the resource.
6.2	Run terraform apply to execute the plan.
6.3	Verify in the AWS console that the new EC2 instance was rebuilt with the new image.
7	**Destroy the EC2 Instance**
7.1	To destroy the EC2 instance, we simply need to run the following command: `$ terraform destroy` Terraform will ask you to confirm your actions before proceeding. Terraform handles all the steps necessary to destroy or remove all resources defined in your workspace. This is a very powerful and destructive feature so please be sure you understand what you are destroying. Also note that it is a good practice to destroy your resources at the end of the day while you are working through this lab guide so you do not incur any additional costs from the service provider, in this case, AWS.

Lab #2: Terraform Variables and Interpolations

Duration 30 minutes

Part A - Variables

Overview

In this lab you will learn how to use both input and output variables as well as interpolations. In the previous lab we used hard-coded values in our configurations which is fine for demo purposes, however in a collaborative development environment we need to find a way to decouple developer and environment specific details out of our configurations.

Skills Learned

In this lab you will learn:

- Use TF_VAR_ to store AWS authentication details in environment variables
- Use a variables file to store compute resource details
- Create a lookup table for regional AWS machine images using a Terraform Map
- Display Terraform 'computed values' using output variables

Instructions

1	**Storing AWS Authentication Credentials in Environment Variables**
1.1	In the first lab we stored our AWS authentication details and our compute resource configuration in the same file. This is a very bad practice. In this lab you will use variables and the power of Terraform variable interpolation to replace hard-coded values with variables.
1.2	Create a new directory for lab2 under the learntf directory you created earlier. `$ mkdir -p ~/learntf/lab2`
1.3	Recall from the lecture that using variables requires us to define and assign values to them. We are going to define our variables in a separate file. Create a new file under lab2 named *'variables.tf'*

1.4	Edit *variables.tf* in your favorite text editor and add the following lines:
	
	`variable "aws_access_key" {}`
	`variable "aws_secret_key" {}`
	`variable "aws_region" {`
	`  default = "us-east-1"`
	`}`
	
	The lines above are used to declare or define the variable. All variables used within Terraform must be defined once. Variables can be defined in any file ending in .tf since Terraform loads all files ending in .tf automatically.
	
	Default values for a variable can be set using the default attribute as shown in the region example above.

1.5	Save the file.

1.6	Create a new file named *provider.tf* under the lab2 directory and add the following lines:
	
	`provider "aws" {`
	`  access_key = "${var.aws_access_key}"`
	`  secret_key = "${var.aws_secret_key}"`
	`  region     = “${var.aws_region}”`
	`}`
	
	Notice how we have replaced a hard coded values with variables. Terraform will interpolate the value at run-time.

1.7	Save the file.

1.8	In a terminal window or shell set the following environment variables:
	
	# For bash shells
	`$ export TF_VAR_aws_access_key=”YOUR_ACCESS_KEY”`
	`$ export TF_VAR_aws_secret_key=”YOUR_SECRET_KEY”`
	
	# For Windows Command Prompt
	
	`C:\> set TF_VAR_aws_access_key “YOUR_ACCESS_KEY”`
	`C:\> set TF_VAR_aws_secret_key “YOUR_SECRET_KEY”`
	
	Be sure to replace with your actual AWS access and secret key from earlier.

2	**Create a Compute Instance in EC2**

2.1	Create another terraform file named compute.tf under the lab2 directory and add the following lines of code:
	
	`resource "aws_instance" "web_server" {`
	`  ami           = "ami-b374d5a5"`
	`  instance_type = "t2.micro"`
	`}`
	
	Remember that AMIs are tied to a specific AWS region. If you are using a region different than us-east-1 then you will need to use a different AMI. Refer to the first Lab in this guide for determining what images are available in a region. You will learn shortly how to use a lookup table to solve this problem.

2.2	Save the file.
2.3	We need to run Terraform init in our lab2 directory since this is an entirely new project. `$ terraform init`
2.4	Go ahead and run terraform plan at this stage. `$ terraform plan` If you forgot or have not properly assigned values for any of your variables, you will be prompted by Terraform at this point to provide those values.
2.5	Go ahead and run terraform apply `$ terraform apply`
2.6	If Terraform executes successfully, then you have successfully used environment variables for your AWS credentials and the default value for the AWS region. If you encountered authentication issues running Terraform, be sure the two AWS environment variables are properly set. Export or setting environment variables directly from within a shell session do not persist from session to session. If you want your environment variables to persist permanently you would store them in your shell's profile file in Linux or as a user environment variable in Windows.
2.7	Destroy the resource using terraform destroy.
3	**Using a Map as a Lookup Table**

3.1	In AWS and most other cloud providers, machine or OS images are usually tied to a specific region, which makes hard coding image IDs a challenge. In this lab you will create a lookup table using a Terraform map. The lookup table will be used to look up an AMI based upon the region we specify.
	
	Recall the syntax for defining a map variable is:
	
	`variable map_name {`
	`  “key1” = “value1”`
	`  “key2” = “value2”`
	`  …`
	`}`
	
	In this lab we will use AWS region codes as keys to lookup AMI image IDs.

3.2	We will define our AMI lookup table in the variables.tf file first.
	
	`variable webserver_amis {`
	`  type = "map"`
	`}`
	
	This definition declares a variable named webserver_amis with a type of map.

3.3	One way to persist variable values is to put them in a file named terraform.tfvars. This file is special in that Terraform automatically loads this file to populate the variables.
	
	Create a new file named terraform.tfvars and add the following code:
	
	`webserver_amis {`
	
	`  # These images are based on RHEL 7.5`
	
	`  # US Northern Virginia`
	`  "us-east-1" = "ami-6871a115"`
	
	`  # US NorthCal`
	`  "us-west-1" = "ami-18726478"`
	
	`  # Mumbai, India`
	`  "ap-south-1" = "ami-5b673c34"`
	
	`  # Frankfurt, Germany`
	`  "eu-central-1" = "ami-c86c3f23"`
	
	`  # Sao Paulo Brazil`
	`  "sa-east-1" = "ami-b0b7e3dc"`
	
	`}`

	In the code above the map variable *webserver_amis* associates a specific AMI with a specific region. As of the time of this publication, these AMIs refer to the RHEL 7.5 HVM image which is part of the AWS Free Tier.
3.4	Save the file.

3.5	Now we need to update our compute resource configuration to use this map. We will do this using a lookup function to look up the value for an AMI based upon the region we specify.
	
	Edit the compute.tf file and update the resource configuration to match the following:
	
	`resource "aws_instance" "web_server" {`
	`  ami           = "${lookup(var.webserver_amis,`
	`                    $var.aws_region)}"`
	
	`  instance_type = "t2.micro"`
	`}`
	
	The lookup function here takes two parameters, the name of the map variable and the lookup or key value.

3.6	Save the file.
4	**Overriding the Region Variable on the Command Line**
4.1	With the changes we just made, our Terraform code now supports creating a compute instance in a number of regions. Let's go ahead and create an EC2 instance in a region other than the default us-east-1 region. To do so we will specify a region on the command line. `$ terraform plan -var 'aws_region=us-west-1'` Recall that the aws_region variable is used in two places: 1) Our AWS provider configuration requires us to specify a region. This tells Terraform which API gateway to hit. 2) Our Web Server configuration uses the region to determine which AMI to use since these are region specific.
4.2	Let's go ahead and apply. Don't forget to specify the region! `$ terraform apply -var 'aws_region=us-west-1'`

4.3	Once Terraform completes, verify the compute instanced was created in the correct region. To do this, log into the AWS Console then select the appropriate region from the drop-down next to your login name.
4.4	Destroy the instance using terraform. Be sure to specify the region here a well otherwise Terraform will default the region and the state file will become inaccurate.
5	**Use Output Variables to Store and Retrieve Data**
5.1	When we run terraform plan or apply to create our EC2 instance, you probably noticed that many of the values for the resource have 'computed' after them. This means that these values will be computed by AWS. Howevter when we run terraform apply, we do not see any of this information returned. Output variables are a way of capturing these computed values and other values as well for printing or querying. The example you are about to walk through will capture the public IP address generated for an EC2 instance you create.

5.2	Edit the compute.tf and add the following lines of code after the compute resource configuration.
	
	`output "webserver_public_ip" }`
	`  value = "$aws_instance.web_server.public_ip"`
	`}`
	
	We use the Terraform variable syntax to get at the public IP attribute for our web server. The format is
	
	`<resource_type>.<resource_name>.<attr>`

5.3	Save the file.
5.4	Run terraform plan
5.5	Run terraform apply
5.6	Verify you see Terraform output a value for webserver_public_ip at the end of its execution. You should see the public IP address for the web server.
5.7	You can also query Terraform after apply to retrieve values as well. `$ terraform output webserver_public_ip`

5.8	Destroy the resource by running terraform destroy.

Part B – Interpolations

Duration 30 minutes

Overview

In the Variables lab we worked with three different types of interpolations: a variable reference, resource reference, and a built in function. In this lab we will working with some of the other types including conditionals for incorporating basic logic into your Terraform, templates for managing long strings or text files, and basic math functions.

Conditionals are typically used for enabling and disabling a component of infrastructure. A use case for templates may be to parameterize a configuration file at deployment time with infrastructure-specific details such as a hostname or IP address or an object storage bucket URL.

Skills Learned

In this lab you will learn how to:

- Use conditionals using the ternary syntax
- Customize strings and text files using inline and file-based templates
- Perform basic math functions

Instructions

1	Conditionals
1.1	Terraform supports conditionals using the ternary syntax which looks like this: CONDITION ? TRUE_VALUE : FALSE_VALUE If you have not see the ternary syntax before, you may be more familiar with the traditional IF THEN ELSE construct in other programming languages. In this lab we will use a conditional to determine whether or not to create a bastion server in our environment. In our example we only need to deploy the bastion server in a production environment. We do not need to create the bastion server in development.

1.2	First create a new variable that will be used to specify which environment we are deploying into.
	
	Edit variables.tf and add the following variable definition:
	
	`variable “target_env” {`
	`  default = “dev”`
	`}`
	
	Save the file.

1.3	Next create a new AWS instance for our bastion server.
	
	Edit compute.tf and add the following configuration:
	
	`resource "aws_instance" "bastion" {`
	`  ami = "${lookup(var.webserver_amis, var.aws_region)}"`
	`  instance_type = "t2.micro"`
	`}`
	
	This code is an exact copy of our web_server, however we have given this instance the name bastion.

1.4	Now let’s add a conditional statement that will specify the number of bastion servers needed depending on the target environment. Add the following line of code inside the bastion resource: `  count = "${var.target_env == "dev" ? 0 : 1}"` The count attribute specifies the number of resources to create. This attribute is optional and is defaulted to 1. In this case however we are using a conditional statement to specify the count value. If target_env is equal to dev, then count will be 0, meaning Terraform will not create this resource. However, if target_env is anything other than dev, then Terraform will create 1 instance of bastion. This is a common example of how to use conditionals to turn on or off features or components using Terraform.
1.5	Save the file.
1.6	Run terraform plan and notice the number of resources Terraform will create. The default value for target_env is dev, so Terraform will not create any bastion instances.
1.7	Set the target environment to ‘prod’ and re-run terraform plan. The target environment can be overwritten on the command line. `$ terraform plan -var target_env=prod`

	Verify that Terraform will create a single bastion server.
2	**Managing multiple instances of a resource**
2.1	In this previous lab we introduced the count attribute which is used to create multiple instances of the same resource. Terraform provides a special variable for accessing these instances elsewhere in your Terraform. In the next example we will create multiple bastion servers then use a special interpolation to access each instance.

2.2	Let's pretend for a moment that we want to create 3 bastion servers.
	
	Update compute.tf and change the quantity of the bastion servers from 1 to 3.
	
	`resource "aws_instance" "bastion" {`
	`  ami = "${lookup(var.webserver_amis, var.aws_region)}"`
	`  instance_type = "t2.micro"`
	`  count = "${var.target_env == "dev" ? 0 : 3}"`
	`}`

2.3	Next we will create an output variable that contains the private IP addresses of all the bastion servers.
	
	Add the following code to compute.tf:
	
	`output "bastion_ips" {`
	`  value = "${aws_instance.bastion.*.private_ip}"`
	`}`
	
	Have a look at the interpolation being used for the output variable value. There is a splat or asterisk inside the interpolation which is positioned right after the resource name. The splat here tells Terraform we want all instances of bastion. The private_ip attribute after the splat tells Terraform that we want private IP addresses of all the bastion instances.

2.4	We can also reference a specific bastion instance using an index. Add another output variable that captures the private IP address of the first bastion server.
	
	`output "bastion_ip_0" {`
	`  value = "${aws_instance.bastion.*.private_ip[0]}"`
	`}`
	
	Notice here in the interpolation syntax we are specifying an index after the attribute. This tells Terraform to return the first element (indexed starting at zero) in the list.

2.5	Save all your changes.
2.6	Run terraform plan to make sure everything looks correct. Be sure to specify production as your target environment, otherwise bastions will not be created.

2.7	Run terraform apply with target environment set to production. Observe the output at the end of the terraform run. You will have two new output variables; one for a single bastion server and one containing a list of private IPs for all bastion servers.
2.8	Run terraform destroy with target environment set to production to remove all resources.
3	**Templates**
3.1	Templates are useful for managing configuration files that contain environment specific details. We can create a template of the configuration file and then specify variables inside the template that will be replaced with real values at runtime. In this lab we will create an IAM policy specific to our web server compute instance that will allow only certain actions to be performed in the AWS console or through the API. The policy takes an ARN or an Amazon Resource Number. We will use a variable for the ARN in our template file. This will allow us to re-apply this policy overtime without having to update the policy with a new ARN every time we create the webserver.

3.2	Create a new file in the lab directory named policy.tpl and add the following code:
	
	`{`
	`   "Version": "2012-10-17",`
	`   "Statement": [{`
	`      "Effect": "Allow",`
	`      "Action": [`
	`         "ec2:DescribeInstances", "ec2:DescribeImages",`
	`         "ec2:DescribeTags", "ec2:DescribeSnapshots"`
	`      ],`
	`      **"Resource": "${arn}"**`
	`   }`
	`   ]`
	`}`
	
	This is an example of an AWS IAM policy. The Resource attribute allows us to specify which resources this policy applies to. The attribute takes an ARN. Here we are using a variable that will be substituted by Terraform.

3.3	The next step is to define the template in Terraform and map the variables to be used for substitution into the template. Edit compute.tf and add the following code: `data "template_file" "webserver_policy_template" {`

```
  template = "${file("${path.module}/policy.tpl")}"

  vars {
    arn = "${aws_instance.web_server.arn}"
  }
}
```

This block of code, known as a data source configuration, defines our policy template and assigns values to the variables that will be used inside the template.

3.4 Templates are rendered by calling the template_file's rendered attribute. We can use an output variable to display the rendered policy. Add the following code to compute.tf.

```
output "web_server_policy_output" {
  value =
"${data.template_file.webserver_policy_template.rendered}"
}
```

Notice here the value of the output variable 'rendered' is the rendered attribute of the data source configuration.

3.5 Run terraform plan and terraform apply.

Terraform apply will display the rendered policy to standard out.

4 Simple Math

4.1 You can perform simple math operations inside of an interpolation. The Terraform console can be used to experiment with interpolations without affecting any existing state.

Launch the console by running the following command:

```
$ terraform console
```

Once the console is launched you should have a prompt

```
>
```

4.2 Try performing some basic math operations.

```
> 1 + 3
4

> “${ 3 * 4 }”
12
```

4.3 You can exit the console by either typing exit or Ctrl-C or Ctrl-D.

Lab #3: Resource Dependencies

Duration 30 minutes

Overview

Terraform makes managing resource dependencies fairly easy. In fact, most of the time you need not worry about the order in which resources are created. Terraform builds a dependency graph when it is executed, which determines which resources are dependent on others and which are not. Terraform is able to parallelize operations based upon this dependency graph, thereby being very efficient at creating resources.

Terraform supports both implicit and explicit dependencies. In this lab you will use an implicit resource dependency to add a web server to the AWS default subnet. You will also use an explicit dependency to create a storage bucket for the web server.

This lab assumes you have a default VPC with a default subnet created. These default networking resources are automatically created for AWS users when they sign up for an account.

If you have deleted your default VPC, please re-create it now by following the instructions below.

https://docs.aws.amazon.com/AmazonVPC/latest/UserGuide/default-vpc.html#create-default-vpc

Skills Learned

In this lab you will learn how to:

- Configure an implicit resource dependency
- Configure an explicit resource dependency

Instructions

1	**Using Implicit Resource Dependencies**
1.1	You define an implicit dependency between two resources whenever an attribute of one resource points to another resource using variable references. Terraform automatically determines the order in which to create

	resources which is why it does not matter what order your configurations are placed in your terraform file or files. In this lab we are going to create a web server that lives in the default AWS subnet. To do this we will define two resources • Public Subnet in a specific Availability Zone • Web Server
1.2	Before we begin, ensure that you have destroyed all existing resources in AWS by running terraform destroy or through the AWS console.
1.3	To get started, create a lab3 directory under the learntf directory created in lab1. `$ mkdir -p ~/learntf/lab3`
1.4	We want to re-use as much as we can from lab2, so copy all the .tf and .tfvar files over from lab2 to lab3. `$ cp ~/learntf/lab2/*.tf ~/learntf/lab2/*.tfvars ~/learntf/lab3`
1.5	Initialize the lab3 directory using terraform init.

1.6	Create a new terraform file named network.tf under lab3 and add the following lines of code:
	
	`resource "aws_default_subnet" "learntf_default_subnet"`
	`{`
	`  availability_zone = "us-east-1a"`
	`}`
	
	The above code will create a reference to the default subnet in a particularly zone. The aws_default_subnet type is special in that it does not create a subnet, merely it creates a reference to the default subnet that already exists. Terraform does not manage default subnets and VPCs. The default VPC and default subnet merely provide a quick way to get up and running with AWS from a networking perspective.
	
	In this example we are using the us-east-1a availability zone. Subnets must be created in these zones which are specific to a particular region. If you specify us-east-1a as the zone, then you must use the us-east-1 as the region, which in this lab, is the default region.

1.7	Save the file.

1.8	Next we want to associate our web server with the subnet we just defined. To do this we must add a reference to the subnet using a variable. This is our first implicit dependency.
	
	Edit the compute.tf file and add the subnet_id attribute to the web_server resource. The value of the subnet_id attribute will be the subnet ID of the default subnet.
	
	`resource "aws_instance" "web_server" {`
	`  ami           = "${lookup(var.webserver_amis, var.aws_region)}"`
	`  instance_type = "t2.micro"`
	`  subnet_id = "${aws_default_subnet.learntf_default_subnet.id}"`
	`}`

1.9	Save the file.
1.10	Now run terraform apply with no arguments. Be sure you have sourced or set your AWS credentials. `$ terraform plan`
1.11	Go ahead and run terraform apply to create the resources.
1.12	Log into the AWS console to view the resources that were created. The VPC and subnet can be viewed under Services > VPC > Your VPCs And Services > VPC > Subnets.
2	**Defining an explicit dependency**
2.1	There are some dependencies that Terraform may not be aware of. For instance there could be application dependencies on a particular resource that are defined at the application level, not at the infrastructure level. An example of this would be a boot strap script that runs on a compute resource that pulls down software from object storage. In this scenario we would want to ensure that the S3 object storage bucket was created first before the compute instance booted.

2.2	Create a new terraform file named storage.tf under lab3 and add the following lines of code:
	
	`resource "aws_s3_bucket" "learntf-bins" {`
	`  # bucket = "learntf-bins"`
	`  acl    = "public-read"`
	`}`
	
	`output "bucket_url" {`

	`  value = "${aws_s3_bucket.learntf-bins.bucket_domain_name}"`
	`}`
	
	This configuration will create a public read only S3 bucket. I intentionally commented out the name of the bucket here since bucket names must be globally unique across all of Amazon. Terraform will generate a bucket name for us.
	
	The last block of code creates an output variable that will display the URL for the public bucket. After the bucket is created you can browse to this bucket in our browser.

2.3	Save the file.

2.4	Edit compute.tf and add an explicit dependency on the new S3 bucket.
	
	`resource "aws_instance" "web_server" {`
	`  ami           = "${lookup(var.webserver_amis,`
	`                      var.aws_region)}"`
	`  instance_type = "t2.micro"`
	`  subnet_id =`
	`"${aws_default_subnet.learntf_default_subnet.id}"`
	
	`  depends_on    = ["aws_s3_bucket.learntf-bins"]`
	`}`
	
	The depends_on attribute is used to define a list of other resources that this compute resource requires.

2.5	Save the file.
2.6	Run terraform plan
2.7	Run terraform apply. Notice that the S3 bucket is the first resource to be created by Terraform, followed by the networking resources and the compute resource. Terraform will display the bucket url value, however you will notice that the public IP for the web server is blank. This is because we created our web server in a private subnet. As a result, no public IP address is assigned.
2.8	You can log into the AWS console to verify the S3 bucket was created under Services > S3 or you can browse to the bucket using the bucket_url.
2.9	Destroy the resources by running terraform destroy.

Lab #4: Terraform Provisioners

Duration 1 hour

Overview

Terraform provisioners are used to perform actions either locally or remotely when creating a resource. Provisioners are most often used to bootstrap a compute instance, which may involve deploying configuration management software, such as chef client or puppet.

In this lab you will configure a provisioner to install and configure apache.

Skills Learned

In this lab you will learn the following:

- Configure ssh keys for the web server compute instance
- Install apache using a remote-exec privisioner
- Configure apache using a file provisioner

Instructions

1	**Create a Lab 4 Working Directory**

1.1	Create a lab4 directory underneath learntf/ and copy all the .tf files and .tfvars from lab3.
	
	`$ mkdir ~/learntf/lab4`
	`$ cp ~/learntf/lab3/*.tf ~/learntf/lab3/*.tfvars ~/learntf/lab4`

1.2	Initialize the lab4 working directory using terraform init.
	
	`$ cd ~/learntf/lab4`
	`$ terraform init`

2	**Configure SSH keys for the web server**
2.1	In order to remotely connect to our web server we must configure SSH keys first and then reconfigure our web server resource to use those SSH keys. The steps for creating SSH keys in linux and windows differ significantly. For Linux:

Open a terminal window and run the following command to generate SSH keys

```
$ ssh-keygen -t rsa -b 4096 -f aws_rsa
```

Do not specify a password on the private key.

This command will create both public and private keys. The private key, aws_rsa, should be protected on your system. The public key, aws_rsa.pub, will be uploaded to the web server.

For Windows users, please follow these directions to generate public and private SSH keys.

https://www.ssh.com/ssh/putty/windows/puttygen

2.2 The next step is to define an AWS key resource in our configuration. This resource will be used to specify our public key.

Create a new file in the lab directory named keypairs.tf and add the following lines of code:

```
resource "aws_key_pair" "deployer-keypair" {
  key_name    = "boostrap-key"
  public_key = “${file(“/path/to/aws_rsa.pub”)}”
}
```

Be sure to update the location of your public key.

2.3 Edit the compute.tf file and add the following line of code to the compute resource:

```
resource "aws_instance" "web_server" {
…
  key_name = "${aws_key_pair.deployer-
keypair.key_name}"

}
```

The key_name attribute specifies the AWS key to use when creating the EC2 instance.

3 Enable SSH/22 Traffic

3.1 Network access in AWS is controlled using security groups and security rules. The default VPC and security group that we are working with by default disallow incoming ssh traffic.

In order to ssh into our web server, we are going to create a new security group with a new rule that will allow incoming traffic on port 22.

To do this, edit the network.tf file and add the following lines of code:

```
resource "aws_security_group" "web_server_sec_group" {
  name        = "web server security group"

  # allow ssh only
  ingress {
    from_port   = 0
    to_port     = 22
    protocol    = "tcp"
    cidr_blocks = ["0.0.0.0/0"]
  }

  egress {
    from_port       = 0
    to_port         = 0
    protocol        = "-1"
    cidr_blocks     = ["0.0.0.0/0"]
  }
}
```

This resource defines a new security group that will be attached to our default VPC. The ingress rules are defined inline in the resource and specify what traffic is allowed in.

The egress block determines what traffic is allowed out of our VPC. In this case we are allowing all traffic to the internet from our VPC.

3.2 In order for security rule to be enforced, resources must be associated with it. To do this we must edit our web server resource.

Add the following line of code to our web server configuration:

```
resource "aws_instance" "web_server" {
 …
  vpc_security_group_ids =
["${aws_security_group.web_server_sec_group.id}"]
}
```

Your web server resource should now look like the following:

```
resource "aws_instance" "web_server" {
  ami           = "${lookup(var.webserver_amis,
var.aws_region)}"
  instance_type = "t2.micro"
```

	`subnet_id = "${aws_default_subnet.learntf_default_subnet.id}"`
	`key_name = "${aws_key_pair.deployer-keypair.key_name}"`
	`vpc_security_group_ids = ["${aws_security_group.web_server_sec_group.id}"]`
	`depends_on = ["aws_s3_bucket.learntf-bins"]`
	`}`

4	**Verify Connectivity**
4.1	Go ahead and run terraform plan and terraform apply.

4.2	Once terraform apply has completed, ssh into our web server.
	
	In a Linux shell:
	
	`$ ssh -i /path/to/aws_rsa ec2-user@52.91.229.240`
	
	Note here that we are using the -i parameter to specify our private SSH key. ec2-user is the default OS user that AWS creates.
	
	You will have to replace the IP address above with the public IP address of the web server.
	
	You can find the public IP value in the output of terraform apply or you can run the following command:
	
	`$ terraform output webserver_public_ip`

4.3	Once you have verified you can log into the web server using ssh, go ahead and disconnect from the web server.
4	**Install Apache using a Provisioner**

4.4	In this section we are going to install Apache remotely by creating a remote execution provisioner. A remote-exec provisioner allows us to execute commands on the web server during the creation process.
	
	Edit the compute.tf file and add the following lines of code inside the web_server compute resource:
	
	`provisioner "remote-exec" {`
	`    inline = [`
	`      "sudo yum install -y httpd",`
	`      "sudo service httpd start",`
	`      "sudo groupadd www",`
	`      "sudo usermod -a -G www ec2-user",`
	`      "sudo usermod -a -G www apache",`
	`      "sudo chown -R apache:www /var/www",`
	`      "sudo chmod 770 -R /var/www"`

	`    ]`
	`}`
	
	The code above defines a remote-exec provisioner. The inline block is merely a Terraform list containing a sequence of commands to be executed. In this case we are installing httpd (Apache) using yum and then starting the apache web server.
	
	We are also setting up some permissions that will allow us to stage some static content later.

4.5	Next we must tell the remote-exec provisioner how to connect to our web server. We do this by configuring a connection resource inside our web server compute resource.
	
	Add the following block anywhere inside the web_server resource.
	
	`  connection {`
	`      type      = "ssh"`
	`      user      = "ec2-user"`
	`      private_key = "${file("/path_to/aws_rsa")}"`
	`  }`
	
	This code block defines how Terraform will connect to our web server. The values here are similar to what we have been using with the ssh client on the command line in Linux.
	
	Make sure you specify the path to the aws_rsa private key you created earlier.

4.6	To access our web server over the internet, we must open up port 80 which is the default http port for apache.
	
	Edit the network.tf file and add the following ingress rule to the web_server_sec_group security group:
	
	`# allow tcp on port 80`
	`  ingress {`
	`    from_port   = 0`
	`    to_port     = 80`
	`    protocol    = "tcp"`
	`    cidr_blocks = ["0.0.0.0/0"]`
	`  }`

4.7	Make sure you save all your changes.
4.8	Destroy all your resources first by running terraform destroy.
4.9	Go ahead and run terraform plan. If all looks correct, run terraform apply.

	During the web server creation process, you should see Terraform remotely connect to the web server, install apache, and start the service.
4.10	After Terraform completes, you should have a running web server that is accessible over the internet. Use your web browser and navigate to: http://WEB_SERVER_PUBLIC_IP Be sure to use the public IP address for the web server. Keep in mind this value will change everytime we re-create the web server. You should be presented with the default Apache home page.
5	**Deploy Files using the File Provisioner**

5.1	The file provisioner allows us to stage files remotely onto our compute resources. This is very useful to pushing configuration files or other static content.
	
	In this section we will create and deploy our own web page to our web server.
	
	In the lab directory, create a file named learntf.index.html and add the following code:
	
	`<html>`
	`  <body>`
	`This is the best web page EVER!`
	`  </body>`
	`<html>`
	
	Save this file.

5.2	Edit the compute.tf and add the following file provisioner after the remote-exec provisioner we just created.
	
	`  provisioner "file" {`
	`    source      = "learntf.index.html"`
	`    destination = "/var/www/html/index.html"`
	`  }`

5.3	Save all your changes and run terraform destroy first, then terraform apply.
5.4	Go ahead and use your browser to hit the public IP address of the web server. You should see your new awesome web page.

5.5 When you are done, go ahead and destroy all your resources.

6 Using Scripts with Remote Execution

6.1 Using inline commands with the remote executioner is fairly simple when you only need to run one or two commands such as creating a group or installing a single package.

However it may be simpler and easier to manage these bootstrapping commands in a shell script that could be used across multiple instances.

Create a new file named bootstrap.sh and add the following lines:

```
#!/bin/sh
sudo yum install -y httpd
sudo service httpd start
sudo groupadd www
sudo usermod -a -G www ec2-user
sudo usermod -a -G www apache
sudo chown -R apache:www /var/www
sudo chmod 770 -R /var/www
```

Note: If you are editing this file in Windows, but sure you configure ‘LF’ for line endings and not ‘CRLF’.

6.2 Save the file

6.3 Edit compute.tf and replace only the remote-exec provisioner block with the following two provisioners:

```
  provisioner "file" {
    source = "bootstrap.sh"
    destination = "/tmp/bootstrap.sh"
  }

  provisioner "remote-exec" {
    inline = [
      "chmod +x /tmp/bootstrap.sh",
      "/tmp/bootstrap.sh"
      ]
  }
```

The first provisioner will upload our bootstrap.sh script and the second provisioner will execute the script.

6.4 You should now have 3 provisioners defined in compute.tf:

```
  provisioner "file" {
    source = "bootstrap.sh"
    destination = "/tmp/bootstrap.sh"
  }
```

	```
	  provisioner "remote-exec" {
	    inline = [
	      "chmod +x /tmp/bootstrap.sh",
	      "/tmp/bootstrap.sh"
	      ]
	  }

	  provisioner "file" {
	    source      = "learntf.index.html"
	    destination = "/var/www/html/index.html"
	  }
	```
6.5	Save the file.
6.6	Run terraform destroy then terraform apply. Terraform will destroy the compute resource first, then re-create it using the new provisioners.
6.7	Once the compute resource is created and bootstrapped, verify you can access the web page using your browser and the new public IP address that was generated.

Lab #5: Data Sources

Duration 30 minutes

Overview

Data sources in Terraform allow you to fetch information on resources that already exist in your infrastructure. These resources may have been created outside of your Terraform scripts.

In this lab you will use a data source to fetch all default VPC subnets from AWS.

Skills Learned

At the end of this exercise, you will learn the following:

- Define an AWS data source
- Use interpolations to access the data source

Instructions

1	**Fetch information using a Data Source**
1.1	Data sources can be used to fetch information about other resources or components that may have been created by someone else outside of your Terraform configurations. One example is fetching an AWS AMI from a list of AMIs that we can then use to create compute instances.
1.2	Create a new lab5 directory under learntf/.
1.3	Copy over the provider.tf, variables.tf, and terraform.tfvars from lab4 to the lab5 directory.
1.4	Create a new file named datasource.tf and add the following lines of code:

```
data "aws_ami_ids" "ubuntu" {
  owners = ["amazon"]

  filter {
    name   = "name"
    values = ["ubuntu/images/ubuntu-*-*-amd64-server-*"]
  }
}
```

This code block defines an AWS data source of type aws_ami_ids. This data source will return a list of AMIs based upon the filters we create. Here we are asking for Ubuntu images owned by Amazon.

1.5 Next we will create an output variable that will print out the entire list of AMIs.

Add the following code to datasource.tf:

```
output "amis" {
  value = "${data.aws_ami_ids.ubuntu.*.id}"
}
```

This output variable will return all the AMI IDs from the data source. Notice here the use of the splat notation between the data source name (ubuntu) and the attribute (id). The splat tells Terraform to return all AMI IDs.

1.6 Save your changes and run terraform apply.

Terraform will simply return a list of Amazon-owned Ubuntu image IDs.

2 Use a Data Source to create other AWS Resources

2.1 In this lab we want to create a couple AWS compute instances across availability zones. We can accomplish this without hard coding values by using a data source to fetch the AZs.

2.2 Create a new file named data.tf and add the following lines of code:

```
data "aws_availability_zones" "available" {}

resource "aws_instance" "webservers" {
  ami = "${lookup(var.webserver_amis, var.aws_region)}"
  instance_type = "t2.micro"

  count =
"${length(data.aws_availability_zones.available.names)}"

  availability_zone =
"${data.aws_availability_zones.available.names[count.index]}"

}
```

The first line of code fetches all the availability zones in the region we configured in our AWS provider.

The rest of the code defines a compute resource where multiple instances are created across each zone. Values from the availability zone data source are used to specify the number of resources to create (count) and the availability zone.

The special variable count.index is used to tell Terraform to essentially create a compute instance in each availability zone.

2.3	Save your changes.
2.4	Run terraform plan then terraform apply.
2.5	You will see Terraform create a compute instance in each zone. Once terraform apply is complete you can log into the AWS console to verify or run terraform show on the command line: `$ terraform show` This command displays the current Terraform state maintained in the state file, which will show each server created and which zone it was created in.
2.6	Destroy all resources by running terraform destroy.

Lab #6: Terraform Modules

Duration 30 minutes

Overview

Terraform modules allow you to neatly package and organize Terraform code for re-use. In this lab we will create resources by using already built Terraform modules hosted in the Terraform Registry.

Most modules define required input variables that must be set when including in your own Terraform.

Skills Learned

At the end of this exercise, you will learn the following:

- Create a VPC using a Terraform module from the Terraform Registry
- Create your own module

Instructions

1	**Create a Simple VPC using Registry Modules**
1.1	In this lab we are going to create an AWS VPC using a module from the Terraform Registry rather than writing our own Terraform.

	Open your browser to https://registry.terraform.io/
1.2	Search the registry for 'aws vpc' and click on the module named 'vpc' authored by AWS.
1.3	The module home page contains documentation, instructions, and usage examples for the module. Take notice of all the networking related resources that this module allows you to create.
1.4	In this lab we are going to create a simple VPC with public and private subnets, including a NAT instance, all using the VPC module from AWS. To get started create the following directory structure under learntf. `$ mkdir -p ~/learntf/lab6/vpc_module`

1.5	Create a new file named vpc.tf in the vpc_module directory and add the following code:
	
	`module "vpc" {`
	`  source = "terraform-aws-modules/vpc/aws"`
	
	`  name = "learntf-vpc"`
	`  cidr = "10.0.0.0/16"`
	
	`  azs             = ["us-east-1a", "us-east-1b", "us-east-1c"]`
	`  private_subnets = ["10.0.1.0/24", "10.0.2.0/24", "10.0.3.0/24"]`
	`  public_subnets  = ["10.0.101.0/24", "10.0.102.0/24", "10.0.103.0/24"]`
	
	`  enable_nat_gateway = true`
	`  enable_vpn_gateway = true`
	`}`
	
	The code above defines a module that we are calling VPC, along with all the required attributes for the module. This module makes it easy for us to create a fairly complete VPC across multiple availability zones, public and private subnets, along with a NAT and VPN gateway.
	
	The source attribute tells Terraform what module to fetch from the Registry. All of the other attributes that follow represent input variables for the module that we are satisfying.
	
	Not all attributes are required. Refer to the Terraform Registry moduel's home page for usage documentation.

1.6	The VPC module does not make any assumptions about the AWS provider, so we must define one ourselves. We will do this in the same vpc.tf file.
	
	Add the following code:
	
	`variable "aws_access_key" {}`
	`variable "aws_secret_key" {}`
	
	`provider "aws" {`
	`  access_key = "${var.aws_access_key}"`
	`  secret_key = "${var.aws_secret_key}"`
	`  region = "us-east-1"`
	`}`

1.7	Next we want to deploy an AWS instance into the VPC. The following code will create an AWS instance in the VPC's public subnet. Take note of how we are using an interpolation to refer to the public subnet ID inside of the module.
	
	`Ter`
	
	Module attributes are referenced using the following syntax (taken from the Interpolations lecture)
	
	`${module.NAME.ATTRIBUTE}`
	
	In the above code, public_subnets is an ordered list, so to retrieve the first public subnet from the list we specify an index of 0 which will return the ID of the subnet.

1.8	Save all your changes.
1.9	Run terraform plan and terraform apply and observe how many resources the VPC module creates for us, including the AWS EC2 instance we defined ourselves.
1.10	You can log into the AWS console to see all the resources that were created.
1.11	Destroy all the resources by running terraform destroy.
2	**Writing your own Module**
2.1	Creating your own modules is fairly simple. Throughout the lab guide we technically have already been creating our own modules called root modules. Modules allow us to organize our code and package it for re-use. A module is a directory containing one or more Terraform files. A module directory may also nest other modules inside.

	In this section we are going to create a very simple module that will be used and referenced from a separate Terraform file.
2.2	Create the following directory under learntf/ `$ mkdir -p ~/learntf/lab6/mod_lab/my_mod` My_mod is where we are going to create our module. The mod_lab directory is where we will reference the module.

2.3	In the my_module directory create a new file named main.tf and add the following code:
	
	`resource "aws_instance" "web_server" {`
	`  ami = "ami-b70554c8"`
	`  instance_type = "t2.micro"`
	`}`
	
	Save the changes. You just created a very simple, but not very useful module.

2.4	Imagine the module we just created is something that was written by someone else and we want to include it in our Terraform code.
	
	Create a new file named module.tf under learntf/lab5/mod_lab and add the following code:
	
	`variable "aws_access_key" {}`
	`variable "aws_secret_key" {}`
	`variable "aws_region" {`
	`  default = "us-east-1"`
	`}`
	
	`provider "aws" {`
	`  access_key = "${var.aws_access_key}"`
	`  secret_key = "${var.aws_secret_key}"`
	`  region = "${var.aws_region}"`
	`}`
	
	`module "my_module" {`
	`  source = "./my_mod/"`
	`}`
	
	The module code at the bottom defines a module named “my_module” and the source attribute specifies where to fetch the module from, in this case the directory my_mod/.

2.5	First run terraform init in the mod_lab directory. This will not only initialize the current working directory but also pull in the my_mod module.

	Try running terraform plan in the mod_lab directory. This will ensure that Terraform can load the module. You do not need to run terraform apply at this time.

2.6	To make the module more modular we can define input variables that can then be used inside the module.
	
	In this lab let us take a simple example and create an input variable that will allow us to specify the number of web servers we want to create.
	
	Edit the main.tf module code and add the following line of code to the web_server resource.
	
	`resource “aws_instance” “web_server” {`
	`  …`
	`  count = “${var.server_count}”`
	`}`

2.7	As with all variables we must declare this variable in our module.
	
	Create a new file named variables.tf under the my_mod directory and add the following code:
	
	`variable “server_count” {`
	`  default = “1”`
	`}`

2.8	Any variable that is declared inside of our module automatically becomes an input variable for that module, which means that it must be assigned a value when you include the module in your terraform code.
	
	Edit the module.tf and add the server_count attribute to the “my_module” module.
	
	`module “my_module” {`
	`  …`
	`  server_count = 3`
	`}`
	
	Here are we setting the server_count to 3 which will cause the module to create 3 web servers.

2.9	Save all files then run terraform plan and confirm that Terraform will create 3 AWS instances.

Lab #7: Terraform State

Duration 60 minutes

Overview

Terraform manages its own state in order to keep track of the resources it manages, and their dependencies. State can be managed locally and remotely using a backend data source.

For a single developer, having Terraform managing state locally is perfectly fine and quite simple. However in a team environment, state needs to be shared and collaborated on from both a development perspective and an operational perspective. In terms of development, resource configuration details can be shared with other team members in a read-only manner. In an operational setting, remote state can be locked to prevent corruption by multiple terraform operations running at once.

In this lab you will work with both local and remote states.

Skills Learned

At the end of this exercise, you will learn the following:

- Import existing infrastructure into Terraform
- Manage local state using terraform show
- Configure a remote backend using AWS S3

Instructions

1	**Import an existing resource into Terraform**
1.1	Terraform allows you to import an existing resource to bring it under the control of Terraform. In this section we will manually create an EC2 instance via the AWS console then import into our Terraform.
1.2	First log into the AWS Console and navigate to Services > EC2.
1.3	Under Instances, click Launch Instances.
1.4	Check the box for Free Tier Only.
1.5	Select any one of the free tier eligible images.
1.6	On the next screen, select the t2.micro type. This should be selected by default.

	Click Review and Launch.
1.7	Review all the details and click the Launch button.
1.8	You will be prompted to provide a keypair for the instance. From the drop down, select 'Proceed without a keypair' and select the Acknowledgement box. Click Launch Instances.
1.9	Once the instance has been launched, copy the instance ID. We will need this ID when importing the compute instance into Terraform.
1.10	The next step is to model this compute instance in Terraform. Create a new lab directory. `$ mkdir -p ~/learntf/lab7`
1.11	Copy over provider.tf, variables.tf, and terraform.tfvars from lab 4 to the new directory.

1.12	Create a new file named compute.tf and add the following code:
	
	`resource "aws_instance" "web_server" {`
	`  instance_type = "t2.micro"`
	`  ami = "ami-759bc50a"`
	`}`
	
	You will have to update the instance_type and ami attributes to match the EC2 instance you manually created earlier in this lab.

1.13	Save the file.
1.14	Initialize the lab7 directory by running terraform init.
1.15	Next we import the resources by running terraform import and associating the ID of the resource with the compute resource configuration: `$ terraform import aws_instance.web_server i-080c6cbc267509872` You will have to replace the resource ID in this lab with the one you just created earlier.
1.16	If the import is successful, the compute instance will now be under terraform's control. To verify this, use Terraform to display the current state. The server should appear in the output.

	`$ terraform state show aws_instance.web_server`
2	**Configure a Remote Backend using AWS S3**
2.1	In this lab we are going to configure Terraform to store state in an S3 bucket. First we need to create a bucket. Log into the AWS console and navigate to Services > S3.
2.2	Click ‘Createbucket’
2.3	In the dialog, provide a name for the bucket and click Next.
2.4	In the Configure Options screen, enable Versioning and Default Encryption. These two options allow us to version our state file and encrypt the state file at rest. Click Next.
2.5	In the Set Permissions screen, click Next.
2.6	In the Review screen, click Create bucket.

2.7	Next we must configure an S3 backend in Terraform. To do this, create a new file named terraform.tf in the lab7 directory and add the following code:
	
	`terraform {`
	`  backend "s3" {`
	`    bucket = "YOUR_BUCKET_NAME"`
	`    key    = "learntf-state/terraform.tfstate”`
	`    region = "us-east-1"`
	`    profile = “default”`
	`  }`
	`}`
	
	Be sure to save the file.
	
	The terraform stanza contains an S3 backend. Here we are specifying the name of the bucket created earlier, along with a key. The key represents where Terraform will store the state in the bucket.
	
	Be sure to replace the bucket name with your bucket name that you created earlier and your access and secret keys.
	
	The profile attribute tells Terraform to load the default profile containing AWS access credentials from a credentials file which we will create in the next step.

2.8	To create an AWS credentials file do the following:

	For Linux users:
	`$ mkdir -p ~/.aws/`
	For Windows users:
	`> mkdir "%UserProfile%\.aws"`
	This will create the directory .aws in your home directory.
	Create a new file in .aws/ called *credentials* and add the following lines:
	`# ./aws/credentials`
	`[default]`
	`aws_access_key_id=YOUR_ACCESS_KEY`
	`aws_secret_access_key=YOUR_SECRET_ACCESS_KEY`
	Replace the values above with your access ID and secret key.
	Save the file.

2.9	For linux users, change permissions on the file so that only you can read it. `$ chmod 600 ~/.aws/credentials`
2.10	We must re-initialize our terraform directory to enable the backend change. Run the init command in the lab7 directory. `$ terraform init`
2.11	Terraform will ask us if we want to migrate any existing state from the local backend over to S3. Enter ‘yes’ to copy over the state. Terraform has now migrated your local state over to S3.
2.12	In the AWS console, you can look at the contents of the S3 bucket to see the state file. Navigate to Services > S3 > YOUR_BUCKET_NAME > *learntf* Be sure to replace *your_bucket* with the name of the bucket created earlier. Verify that a state file exists in the bucket.
2.13	From a command line, run terraform show.

	`$ terraform state show` This command displays the current terraform state. We ran this command earlier when importing a resource. During that time, state was maintained locally. Now when we run this command, terraform is fetching the state from S3 object storage, not local storage.
2.14	Destroy the imported resource. `$ terraform destroy` Terraform will destroy the imported resource and update the state file.
2.15	Run terraform state show to confirm the state file was updated.
3	Switch Back to Local State
3.1	You can switch back to local state simply by removing the S3 storage backend from your terraform configuration. Edit the terraform.tf and comment out the entire terraform stanza using /* */ notation.
3.2	Save the file and re-initialize the directory.
3.3	Terraform will ask whether you want to migrate your state file from S3. Enter 'yes' and hit enter.
3.4	Verify Terraform migrated your state file to the lab7 directory.
3.5	Run terraform destroy to destroy any resources created by Terraform.

Extra - Using an AWS Credentials File with Terraform

Rather than use environment variables to store your AWS access and secret key, you may also use a credentials file containing AWS credentials. The credentials file typical remains in your home directory and has the advantage of support multiple profiles or accounts. This is helpful when you may have your own named user credentials but also want to support credentials for different roles such as administrator, operator, or deployer as an example.

One other benefit to the credentials file is that the AWS SDK and CLI also support using the credentials so it provides a single source for secrets.

Lastly, the environment variables need to be set every time you load or reload your shell, while the credentials file is persistent.

To create a credentials file and use it with an AWS Terraform provider, follow the steps below:

1	**Using a Credentials file with the AWS Provider**

1.1	In your home directory either on Linux, OSX, and Windows, create the following directory
	
	Linux
	
	`$ mkdir ~/.aws`
	
	On Windows
	
	`> mkdir "%UserProfile%\.aws"`

1.2	Create a new file named credentials and save it in .aws.
	
	Add the following code:
	
	`# Default Profile`
	`[default]`
	`aws_access_key_id=`*`YOUR_ACCESS_KEY`*
	`aws_secret_access_key=`*`YOUR_SECRET_KEY`*
	
	The above file configures a profile named default. If you already have a default profile defined, you may specify a different profile name here. AWS allows you to support any number of profiles.

1.3	Next we must configure our AWS provider to use the credentials file. Here is the code for the AWS provider:
	
	`provider "aws" {`
	`  region                    = "${var.aws_region}"`
	`  profile                   = "default"`
	`}`
	
	In the provider above we have removed the environment variables for the AWS credentials and replaced them with a profile attribute.
	
	You may now use this provider configuration with any of the labs instead of having to set environment variables for the access and secret keys.

References

Download Terraform
https://www.terraform.io/downloads.html

Terraform AWS Provider Reference
https://www.terraform.io/docs/providers/aws/

Terraform Registry
https://registry.terraform.io/

AWS Free Tier
https://aws.amazon.com/free

Change Log

Author	Summary	Version
Christopher Parent	Initial revision	1.0