Course Overview
Course Overview

Hi everyone my name is Ned Bellavance, and welcome to my course Terraform - Getting Started, an introductory course into the world of Terraform. Terraform is a tool used to automate the deployment of infrastructure across multiple providers in both public and private clouds. Within this course, you will learn the fundamentals of how Terraform functions, as well as work with a real world example to gain hands-on experience with the tool. We will be focusing on a few topics within the course, including creating a basic configuration and updating it with new resources; understanding Terraform components like variables, provisioners, providers, and more; integrating multiple providers in a single configuration; and using abstraction and reusable components such as modules to make your configurations consistent and repeatable. By the end of this course, you will be able to dive into the world of automating infrastructure with Terraform by your side, enabling you to be more productive and do more with less. It's not necessary to know anything about coding or programming prior to starting the course. This is a getting started course, after all. The demonstrations utilize AWS, so a passing familiarity is recommended, but certainly not required. If you're interested in automating infrastructure with one of the most exciting tools out there, I invite you to dive into the Terraform - Getting Started course on Pluralsight.

What You Need to Know About Infrastructure as Code

Overview

Hey, everyone! Welcome to my course, Terraform- Getting Started. Before we dive into the wonderful world that is Terraform, first I wanted to level set a little bit about infrastructure as code. Throughout the course, we are going to be using a lot of terminology that comes from the concepts in infrastructure as code, and in order to get the most out of this course, I first want you to have a good idea of what I mean when I say infrastructure as code. Hey everyone, I'm Ned Bellavance. I'm a HashiCorp Ambassador and founder of Ned in the Cloud. Let's get into what you need to know about infrastructure as code. All right. In this module, we are first going to define infrastructure as code. Got to start with the definition, right? Need to start with first principles. What are we even talking about? And then we'll get into the core concepts of what infrastructure as code is, and then finally, we'll talk a little bit about the benefits of using infrastructure as code. Why would you even use this crazy thing? But first, a definition.

Infrastructure as Code Defined

Infrastructure as code is provisioning infrastructure through software to achieve consistent and predictable environments. There are a lot of important terms in this definition. A few that I want to call out is the fact that this is being done through software. It's not a manual process. And the goal is to achieve consistency. That means every time you use this software to deploy infrastructure, it does it in a consistent way and that the environment you get at the end is a predictable environment. It doesn't leave you guessing. It's going to look exactly like the configuration files say it should look. That's very important, especially when you have

multiple environments that will be running the same version of an application. To achieve this goal, there are some core concepts I'd like to talk about. The first, and this should be fairly obvious, but I feel like I need to say it anyway, is infrastructure as code as defined in code. You're going to be creating files using some sort of software and coding mechanism to define your infrastructure, and whether that format is JSON, YAML, or HashiCorp configuration language, infrastructure as code is going to be defined in code. The next big concept is you should be storing that code somewhere in source control. You are using code, after all. You might as well treat it like code. The source control management that most people are familiar with is Git and GitHub. GitHub uses Git source control to store repositories of code. The code is versioned, and multiple developers can work on it simultaneously. Your infrastructure that you've defined in code should be stored in a versioned source control repository. When it comes to the actual code itself, there are two different approaches to implementing infrastructure as code. There is declarative or imperative. Let's dive into what I mean when I say declarative or imperative with a fun example.

Declarative vs. Imperative

So, I love tacos. My favorite day of the week is Taco Tuesday, and if I were to instruct software to make me a taco in an imperative way, I would do that by telling it what the exact steps are to make me a taco. First get me the ingredients for a taco. You need to get the shell, the beans, the cheese, the lettuce, and the salsa, or at least that's what I like in my taco. And then I need to tell the software how to assemble those ingredients to make me a taco. I would tell the software to put the beans in the shell, put the cheese on the beans, put the lettuce on the cheese, and the salsa on the lettuce, because that is the proper order that one should assemble a taco in. Now you can see this is very procedural in nature. I'm telling the software exactly what to do. Declarative takes a slightly different approach. Now let's say in a declarative world I also want software to make me a taco. That software is going to have a rudimentary idea of how to make food, kind of like a cook. Just like I can tell a cook I want a taco with the following toppings, I can use a configuration language like HashiCorp Configuration Language to declare what I want. In this case, I'm telling it I want something that is of type food and of the sub-type taco, and I'm going to give it a name I can refer to it with, in this case bean-taco. And then within my configuration block, I'm going to tell it the ingredients I want in my taco; beans, cheese, lettuce, and salsa. And that's all I have to tell the software. It already has a predefined routine for how to get ingredients and it has a predefined order in which to assemble those ingredients. If I want to change the defaults, I might put additional information in this configuration block, but the idea here is I'm declaring what I want. I want a bean taco with these ingredients, and then I'm leaving it up to the software to figure out exactly how to implement what I want. Terraform is an example of a declarative approach to deploying Infrastructure as Code.

Idempotence and Consistency

Another core concept is idempotence and consistency. You're probably already familiar with the idea of consistency. Each time you do something, the results should be the same. But idempotent is one of those words that gets thrown around, but you may not necessarily know what it means. Let's use another example to define it better. Let's say my niece, who also

loves tacos, has asked me to make her a taco, and I do it. I say, "Here's your taco." Now, in an idempotent world, if she asks me again to make her a taco, I will go, "Um, you already have a taco." I'm not going to go ahead and make another taco because I'm aware of her state and the fact she already has a taco. If she gives me the same instruction again, I'm not going to do anything because her instruction already matches the state of the world she wants. She has the taco. In a non-idempotent world, each time she told me to make her a taco, I would make and give her another taco. Terraform attempts to be idempotent in the sense that if you haven't changed anything about your code and you apply it again to the same environment, nothing will change in the environment because your defined code matches the reality of the infrastructure that exists. And that's what's meant by idempotent

Push or Pull

The last concept to look at with Infrastructure as Code is are you pushing or pulling configurations to the target environment. So again, to give a fun example of what I mean by push or pull, in a push-type scenario, once my niece has expressed her desire for a taco, I would simply go, take this taco, and push the taco over to her, and hopefully, if she's feeling polite, she'll say thanks. In a pull-type scenario, once she expresses that she wants the taco, she'll take the taco from me and I'll say, sure, here you go. In the world of Infrastructure as Code, Terraform is a push-type model. The configuration that Terraform has is getting pushed to the target environment. An opposite example would be a situation where there's an agent running in the target environment and it pulls its configuration from a central source on a regular basis.

Benefits of Infrastructure as Code

Now, all of this is great, and maybe you're feeling a bit hungry right now, but first let's talk about the benefits of using infrastructure as code. Why would you go through all the trouble of defining your infrastructure in code as opposed to just manually going out and deploying it? One, you've automated your deployment, which means that you don't have to go through the manual steps every time you need to build a new environment. That makes deployment faster, and faster is usually better in the world of technology. You've also created a repeatable process. Each time you need to build out or update the environment, you simply apply the configuration. Your new repeatable process can also be used to create multiple consistent environments. This is especially important if you want your dev, QA, staging, and production environments to all match. Your reputable process is defined in code, and code can be reused. Once you figured out how to properly deploy, say, a database server for a particular application, you can take the code for that database server deployment and reuse it in any other application that needs a similar database server backend. Having those reusable components will make your life a lot easier. It follows a principle that developers call Don't Repeat Yourself, or DRY programming. Once you write the code for a process, then you should make that process reusable so you don't have to repeat yourself. Lastly, one of the great things about defining your infrastructure as code is you've actually documented your architecture in the process of defining it within code. I've encountered situations where I thought I understood an architecture, but when I went to go define it as code, I realized there were components that I either didn't realize were part of the architecture, or I didn't

understand how they actually worked. By defining my infrastructure deployment with code, I now had a deeper and better understanding of how my architecture was actually working. That's a huge benefit when everything is documented in code.

Summary

Hopefully, in this first introductory module, I've allayed some of your concerns about Infrastructure as Code. It really isn't all that scary, and it does make your life easier. At the end of the day, manual processes are generally the enemy. Humans are fallible, we make mistakes, we forget things all the time, and if you are relying on manual processes to deploy environments consistently and repeatedly, at some point someone's going to miss a step; it's just the nature of human beings. Automating your infrastructure deployments makes a lot of sense. Lastly, when in doubt, go have a taco and think about it. I find that walking away for a moment on a particularly difficult project and having something delicious, like a taco, really helps my thinking process. And now, I kind of want a taco too. Coming up in the next module, we're going to dive into Terraform proper by deploying our first Terraform configuration, so I hope you'll join me there.

Deploying Your First Terraform Configuration

Overview

Now that we've laid a solid foundation for what Infrastructure as Code is, it's time to dive into Terraform, and I find the best way to do that is to go right in and get something deployed so you can start getting your head around the core concepts that make up Terraform. That's what we're going to do in this module. Hey, everyone. This is Ned Bellavance. I'm HashiCorp ambassador and founder of Ned in the Cloud, and this is Deploying Your First Terraform Configuration. All right, what are we going to cover in this module? Well, first, we're going to talk about what Terraform even is. You probably already have some idea, but now we're going to get into the core components that make up Terraform, the basic workflow you'll use with Terraform to deploy infrastructure, and how to get Terraform installed on your workstation so you can follow along. Before we dive into actually deploying a Terraform configuration, I'd like to present you with a real-world scenario that's going to help place some of the tasks and information you'll be learning into a real-world context. I know as an IT practitioner I'm always eager to see where the rubber meets the road with any new tool, and I suspect you're going to feel the same way, and having a real-world construct where you would be practicing these skills helps make sense of what might be just abstract concepts. And then finally, we're going to walk through a demonstration of deploying a basic configuration based on the requirements we define in the scenario. You can think of it as a Terraform: "Hello world", if you like, but I promise it's going to be a little more useful and practical than your usual hello world example.

What Is Terraform?

Let's explore what Terraform is, the core components that are used with Terraform, and how to get it installed. Terraform is simply a tool to automate the deployment and management of infrastructure. The term infrastructure can be a bit nebulous, but I like to think of it as any layer of technology that a developer consumes without having to deploy and manage it. Networking, virtual machines, even containers all fall under the moniker of infrastructure. The core of Terraform is an open-source project maintained by HashiCorp. There are paid versions of Terraform available as either Terraform Cloud or Terraform Enterprise. We're not going to cover those services in this course. We'll be sticking with the core open-source version only. Terraform is also a vendor agnostic, meaning it doesn't prefer any particular cloud or service. You can use it against AWS, Azure, DigitalOcean, VMware, etc. Pretty much any infrastructure service you can think of probably works with Terraform The core software for Terraform is a single binary compiled from Go. HashiCorp offers compiled versions for multiple operating systems, so chances are there is a Terraform binary that will work for you. As I covered in the previous module, Terraform configuration files use a declarative syntax rather than an imperative one. You are describing how you want the world to be, and Terraform is in charge of handling the heavy lifting. The actual configuration files are written in either HashiCorp Configuration Language, a derivative of JSON, or in JSON directly. Unless you are using another programming language to create Terraform configuration files, I'd recommend sticking to HCL. It's much more human readable and human writeable. Finally, Terraform uses a push style of deployment to create infrastructure. Terraform is going to reach out to the API for any given service and tell it what to create. There's no agent to install on a remote machine. That's a relief for those of us who have developed agent fatigue over the years. One less thing to patch and maintain is a positive in my book. There are four core components you should be aware of in Terraform. The first is the executable itself. This is the single binary file you invoke to run Terraform. It contains all the core Terraform functionality. The configuration that you're going to deploy will be contained in one or more Terraform files, which typically have the file extension .tf. When Terraform sees one or more Terraform files in a directory, it will take all of those files and stitch them together into a configuration. The next component is how Terraform talks to all the various services out there. The provider plugins are executables invoked by Terraform to interact with a service's APIs. For instance, AWS would be considered a provider, and if Terraform wants to talk to AWS and provision resources, it uses a provider plugin to do so. The most common plugins are hosted on the public Terraform Registry at registry.terraform.io. And then finally, once resources have been created, Terraform likes to keep track of what's going on, so it maintains state data which contains the current information about your deployment. It's a mapping of what you've defined in your configuration to what exists in your target environment. When you want to do an update of your environment, Terraform compares your updated configuration to what is in the state file, calculates the changes needed to make the two match, and then makes the changes and updates the state data.

Installing Terraform

Installing Terraform is exceedingly simple. You simply download the executable compiled for your operating system, make sure that it's added to your path variable, and start using Terraform. Terraform is also available in common package managers like apt,

yum, Homebrew, and Chocolatey. You could even grab it as a Docker container. Why don't we go check it out? Woohoo, it is demo time y'all! In this demonstration, we're going to run through a couple quick items. First I'll show you where you can get Terraform installed. Once you've got it installed, we can try out some of the basic commands so you can learn the command structure favored by Terraform. If you would like to follow along, and I hope you do, you're going to need a system where you can install Terraform, a code editor to view the files, and the actual exercise files themselves. You can find those by going to the Exercise files tab in the course and following the link to my GitHub repository for Terraform - Getting Started. I like to keep the exercise files on GitHub so I can keep them up to date, and folks like you can file issues if you find them. Now let's jump over to the demo environment and get started.

Using the CLI

All right, here we are in Visual Studio Code. This is my preferred code editor of choice, but you use whatever works for you. I like this because I can see all of my files in the left pane, I can see the contents of those files in the center pane, and I can bring up a terminal from the bottom if I want to run commands all from within Visual Studio Code. I have the exercise files open in the left pane, so let's go over those very quickly. Again, if you're looking for these exercise files, you can find them by going to the Exercise files tab or going to my GitHub repository. In the top folder base_web_app, we have the base configuration that we'll be working with, and it's called main.tf. In the commands folder, we have the commands that you can run for each of the modules in this course. Then below the commands directory, we have a directory for the solution for each module beyond module three. Now don't worry about what's in those right now. We'll discuss that more when we get to module four. For now, let's expand commands and open up m3_commands. Now if you want to play along, the first thing you're going to need to do is install Terraform if you don't already have it, and you can get it if you go to terraform.io/downloads. Let's take a look at that page right now. Here is the download page for Terraform. If we scroll down a little bit, we can see all the different operating systems and the downloads for each of those operating systems. This would be one way to install Terraform. If we scroll up a little bit, we can see that there are instructions for setting up the repository for APT or Yum if you wanted to install it that way. If you're using macOS, you can use Homebrew, and if you're using Windows, you could use Chocolatey to install Terraform. Let's head back to Visual Studio Code. Now I'm running Windows, so I used Chocolatey to install Terraform. Let's go ahead and bring up the terminal and see what version of Terraform I'm using. All right, first, I am going to run terraform version, and I am running Terraform version 1.0.8. That is the same as what we saw on the download page, so I'm running the current version on Windows AMD 64. And if I needed to upgrade my Terraform, I could run choco upgrade terraform -y, and that would upgrade my version, but I'm on the current one, so we're good. Now if we want to get some information about how to use the Terraform CLI, we could run either terraform -help or just type in terraform. The output from just running Terraform will list out the main commands you'll use, as well as other commands that are available. If we look at the general usage for the CLI, we can see it's terraform, followed by any global options, then the subcommand that you want to run, followed by arguments. If we scroll down to the bottom again, we can see the global options include things like -chdir to specify what directory to run these commands from, -help can be used to get more information about Terraform or a specified subcommand, and -version is an alias for the version subcommand. One other thing I want to point out is when

you're specifying arguments with Terraform, even if the argument is multiple characters, you can still use a single dash instead of a double dash. Terraform will accept either, but the preferred syntax is a single dash. Now we have a base configuration in the base_web_app directory, but before we look at that base configuration, let's first get some context by introducing our real-world scenario.

Globomantics Scenario

To help put some context around what we're going to be doing in this course, I have a scenario involving the fictional company Globomantics. For our real-world scenario, you have just started as an ITOps admin at Globomantics, a global risk assessment company. Congratulations! And welcome to the team! They're excited you're here and they already have a project lined up for you to work on. Your friend, Sally Sue the developer, has requested that you provision a development environment that's going to be part of a new line-of-business application Globomantics is developing to turn their existing product into a SaaS product for their clients. Now the application is a basic web application right now. It's got a web frontend that will serve up content to potential customers. It's nothing super complicated. Globomantics has recently started using the public cloud for deploying its new applications, and you've been asked to spin up this environment in AWS, Amazon Web Services. You could, of course, simply log into the AWS console and set up the environment manually, but someone told you about this new software called Terraform, and this seems like an ideal project to take Terraform for a test run. In fact, Sally Sue has found a really basic Terraform deployment file she thinks you could get started with. The base configuration Sally found includes the following. We're going to be deploying to the AWS us-east-1 region, and within that region we are creating a VPC with a single public subnet. And inside that subnet, we are creating a single EC2 instance that is running Nginx as a web server. We're also going to have to create routing resources and a security group to allow web traffic to reach that web server. You think this sounds like an excellent start. Before we dig into the configuration file, let's first talk a little bit about HCL syntax so you know what you're looking at.

Terraform Object Types

Before we look at the configuration, there are three Terraform object types you need to know about. They are providers, resources, and data sources. Provider blocks define information about a provider you want to use. For instance, we are going to be using the AWS provider and that provider wants to know what AWS account and region you're going to be using. Resources are things you want to create in a target environment and they are the bulk of what you'll be writing. Each resource is associated with a provider and will usually require some additional information for a configuration. A resource could be an EC2 instance, a virtual network, or even a database. Data sources are a way to query information from a provider. You aren't creating anything, you're simply asking for information you might want to use in your configuration. Just like resources, data sources are associated with a provider. A data source could be a current list of availability zones in a region, an AMI to use for an EC2 instance, or a list of templates on a vSphere cluster. Now, what do these object configuration blocks look like?

General Block Syntax

HashiCorp configuration language uses block syntax for everything in the file, it's a simplified version of JSON that is easier to read and it supports inline comments. Each block is going to start with the block type keyword that describes what type of object is being described in the block. Next is going to be a series of labels that are dependent on what type of object we're working with. The last label in the series is usually the name label, which provides a way to refer back to the object in the rest of the configuration. Within that block, we are going to have one or more key value pairs that make use of available arguments for the object type. Each key will be a string and the value could be any of Terraform's different data types, which we'll get into in a later module. You can also have nested blocks inside of the main block. Nested blocks will start with the name of the nested block and curly braces. Inside the nested block will be more key value pairs. This might seem a little too abstract so let's see how the syntax would be applied to an EC2 instance in AWS. The object type we're describing here is a resource. We're creating an EC2 instance so we use the keyword resource, the type of resource is an EC2 instance, which based on the documentation for the AWS provider, uses the label aws_instance. Finally, the name label for our resource is web_server. This gives us a way to refer to it, especially if we've got multiple EC2 instances in our configuration. Inside the block, we can specify a name for our EC2 instance. This is the name that we will see in the AWS console. Finally, we could use a nested block to specify an ebs_volume to attach to our EC2 instance, and inside that block, we could specify the size of the ebs_volume we want. If we have multiple ebs_volumes to attach, we can repeat the nested block multiple times. I've mentioned a few times the ability to refer to other objects inside of a Terraform configuration. HCL has a defined syntax for doing so. The general format to refer to a resource is the resource type, the name label, and then the attributes you want to reference from the resource. If you want the whole resource, you can skip the attribute. As an example, let's say we want to reference the name of our web server. The syntax would be the resource type aws_instance, the name label, web_server, and the attribute, name. By doing this, we can get the value that is stored in the name attribute of our web server. Now that we have a little background about reading HCL syntax, let's take a look at that base configuration.

Reviewing the Base Configuration

All right, let's go ahead and open up the configuration that's stored in the main.tf file. There we go. Comments in HCL are supported by using the pound sign, and in this file, we've used comments to break up the file into providers, data, and resources. Let's first look at the provider block. In our provider block, we're using the provider keyword to say this is a provider object, and then we're specifying the type of provider, in this case, AWS. This will let Terraform know we're using the AWS provider. Inside of the block, we have a set of key-value pairs. We're telling the provider what AWS account we want to use and how we are going to access it by specifying our access key and secret key. And we're also telling it what region we want to use by specifying the argument region and setting it equal to us-east-1. If we scroll down into the data area, we have a single data source here. We use the data keyword to specify that it is a data source. The data source type is aws_ssm_parameter. So, this is a service manager parameter, and we're giving it a name label of AMI. Within the configuration block, we have a single argument name, and we're setting it equal to a path to a parameter. This particular parameter grabs the latest Amazon Linux to AMI ID for the region

we're currently using. We will make use of this value later when we create our AWS instance. Scrolling down a little bit more, we get into the resources portion of our configuration, and we start with networking. We're going to create an AWS VPC, and we start the block by specifying the resource keyword followed by the resource type, in this case, aws_vpc, and then we're giving it a name label of vpc. Inside of the configuration block, we're setting the CIDR block that should be used by the VPC, and we're also enabling DNS hostnames. Looking at the next block, we are going to create an aws_internet_gateway, and we want to associate that internet gateway with the VPC we just created. To do that, inside the configuration block, we have the single argument vpc_id, and then we're using the reference syntax to reference the ID of our VPC. So that is aws_vpc.vpc, because that's the name label we assigned to our VPC resource, and then .id is the attribute that we want from that resource. Now you might be wondering, how do I know what arguments and attributes are available for a resource? And the short answer is you have to read the documentation for the provider and the resource. The longer answer is what we're going to explore in a future module as we add additional resources to this configuration. Scrolling down a little bit more, we can see that we are creating an aws_subnet with the name label subnet1. We're assigning it a cidr_block, and we're referencing the same vpc_id that we just used for the internet gateway, and we're setting map_public_ip_on_launch to true, so when we spin up an EC2 instance in this subnet, it gets a public IP address. Scrolling down a little bit more, we are going to create an aws_route_table called rtb. We're going to associate it with our vpc, and here is our first nested block. In our nested block, we can specify a route to add to that route table. In this case, we're creating a default route and pointing it at our internet gateway. In this way, traffic can get out of our VPC through that internet gateway. The last portion of the networking is associating our route table with our single subnet, and we will do that by creating an aws_route_table_association called rta-subnet1. Within that configuration block, we're going to specify the subnet_id of our single subnet, and the route_table_id of the route table we just created, and now there's an association between those two objects. Scrolling down a little bit more, we are going to create an aws_security_group that allows port 80 from anywhere to talk to our EC2 instance. We are associating this security group with our VPC, and we're creating a single ingress group using a nested block, and inside of that ingress nested block, we're setting the from_port and to_port to port 80 to allow port 80 in, we're setting the protocol to tcp, and the cidr_block is set to all 0's /0, which means allow traffic from anywhere on port 80. And then below, that we have an egress block, and this egress nested block allows outbound traffic to anywhere. Lastly, we have our EC2 instance. We're creating a resource of aws_instance type and naming it nginx1. For the AMI ID, we are now going to be referencing our data source, and we can see the syntax for that is a little different than regular resources. We first have to specify it is a data source by saying data dot the type of data source dot the name label and then the attribute that we want from that data source, in this case, value. So this will return the AMI ID for Amazon Linux 2 in the region we're currently working in. If you're curious about what the non-sensitive term is, that is a function, and we're going to cover functions a little bit later, so don't worry about that for now. Instance_type sets the instance type to t2.micro. We are trying to keep this thing as small as possible to stay on the free tier. The subnet_id will reference the single subnet that we have created, and then the argument vpc_security_group_ids, you see that's plural, that's expecting a list of security group IDs. We only have a single security group ID to give it, but we still need to put it in a list. Lists are enclosed in square brackets, and then the elements in the list are separated by commas. We only have a single element for the list, which is the security group we created to allow port 80. And then lastly, we are sending

some user data to our instance, and this is simply a script that will run when the instance starts up for the first time. In the script, we are installing nginx and starting it up, we're deleting the default index.html file, and replacing it with something else. If you're not familiar with the EOF syntax that you're seeing right there, that is a way of specifying a block of text that should not be interpreted in any way; it should just be passed directly to the argument as is. And this is an easy way for you to specify a script without having Terraform try to interpolate it. The syntax is simply two of the less signs followed by a keyword, in this case, EOF, the text you want, and then closing it with that same keyword, EOF again. That is everything that's in the configuration. Now we need to deploy our configuration. But how do we go about doing that?

Terraform Workflow

Terraform has a basic workflow that allows you to provision, update, and remove infrastructure. Let's dig into that workflow now. If you'll recall from earlier, Terraform makes use of provider plugins to interact with services like AWS. Before it can use those plugins, it needs to get them. This is done as part of the initialization process, and the command to do so is terraform init. Terraform init looks for configuration files inside of the current working directory and examines them to see if they need any provider plugins. If they do, it will try and download those plugins from the public Terraform Registry, unless you specify an alternate location. Terraform will also need to store state data about your configuration somewhere. Part of the initialization process is getting a state data back end ready. If you don't specify a back end, Terraform will create a state data file in the current working directory. Once initialization is complete, Terraform is ready to deploy some infrastructure. The next step in the workflow is to plan out your deployment with terraform plan. In this case, Terraform will take a look at your current configuration, the contents of your state data, determine the differences between the two, and make a plan to update your target environment to match the desired configuration. Terraform will print out the plan for you to look at, and you can verify the changes Terraform wants to make. You don't have to run a terraform plan, but it is pretty useful to know what Terraform is planning to do before it does it. You can save the plan changes to a file and then feed that back to Terraform in the next step. It's now time to actually make changes in the target environment, and you do that by running terraform apply. Assuming you ran terraform plan and saved the changes to a file, Terraform will simply execute those changes using the provider plugins. The resources will be created or modified in the target environment, and then the state data will be updated to reflect the changes. If we run terraform plan or apply again without making any changes, Terraform will tell us no changes are necessary since the configuration and the state data match. There is one more command I want to bring up, which might seem a little strange, and that's terraform destroy. If you are done with the environment, the command terraform destroy will do exactly that, destroy everything in the target environment based off of what is in state data. This is a dangerous command, and Terraform will ask you if you're sure. We're going to use this command in the course to save money when we're done with the module, but in the real world, please take care. Here there be dragons.

Deploying the Base Configuration

With the basic workflow fresh in our brains, let's get our base configuration deployed. We'll start by initializing the configuration, then we'll plan our deployment, and finally, we'll apply the plan to create resources. If you're following along, and again, I hope that you are, you're going to need an AWS account and AWS access keys. I'd recommend creating a separate AWS account to use for this course so it doesn't conflict with anything else, but that's entirely up to you. Quick disclaimer. Some of the resources deployed in AWS may cost you money. I tried to use the smallest instances possible, but there is a chance you will be charged some small amount of money for what you're provisioning in AWS, so consider yourself suitably warned. Alright, let's get our basic configuration deployed. Now before we do that, let's make a copy of our base configuration and edit that copy. So first I'm going to open up the terminal. I'll go ahead and do that now. Alright, and we are going to create a directory called globo_web_app and copy our main.tf file over to that directory. So I'm going to copy those two commands now, and paste them down below. There we go, I have successfully created a directory and copied the main.tf file to it. So we can see that over here on the left. There's the main.tf file. And let's switch to that directory so that we can work in that directory with Terraform. Now before we run through the actual workflow, there is one tiny change we need to make in the main.tf file. Go ahead and open that now. You can see the AWS access_key and secret_key have placeholders in them. I'm going to update those values with a valid AWS access key and secret key. Now while I am filling this out, I want to provide a quick disclaimer. You should never hardcode your access key and secret key into a Terraform configuration. We're doing it right now because we haven't yet learned a better way of doing it, but rest assured, in the next module we are going to remove this from the configuration and never do it again. This is purely for demonstration purposes. In fact, I've already invalidated this access_key and secret_key by the time you watch this video. With that being said, I'll go ahead and save this file, and now we can run through the initialization process, and I'll do that by running terraform init. It's going to go ahead and initialize the backend it will use for state data and download any provider plugins that it needs for our configuration. And lastly, it will create a special lock file called .terraform.lock.hcl. So if we scroll up a little bit, we can see where it initializes the backend, initializes the provider plugins by downloading the latest plugin from the public Terraform registry, and creating that lock file as the last thing. And if we look over in our globo_web_app directory, we can see there's that .terraform.lock file, and there's also a new directory called .terraform, and inside that, if we expand it, that is where it downloads the provider executable that will be used to talk to AWS. Now that our Terraform configuration is initialized, we can go ahead and run terraform plan. So I will run terraform plan, and I'm going to add a new argument here, -out. This will write the plan out to a file, and I'm specifying the file as m3.tfplan. So I'll go ahead and run that now. And as part of the plan, it is going to reach out to AWS and determine what it needs to create to match our AWS environment to what's in the configuration. And that ran pretty quickly. We can see that it's saying in the plan there are seven resources to add. And if we go ahead and expand this all the way up, we can scroll up and review what's in the plan. So let's scroll up to the top, and we can see it starts with the instance that's going to be created. You should note, anything with a green plus sign indicates that the resource or attribute is going to be created. So we can see there is our AWS instance. If we scroll down a bit more, we can see the internet_gateway, the route_table, etc. So it's going to create seven resources in total. It's not going to change any, and it's not going to destroy any. Now if we want to apply our plan, we can simply run terraform apply and feed it our file, m3.tfplan. So I'll go ahead

and do that now. There we go. And if we had run terraform apply without specifying a plan file, it would first print a plan of what it's going to do and then ask for confirmation of the changes that it's going to make. Because we supplied a tfplan file, it doesn't have to confirm those changes because it assumes we've already reviewed that plan. So this could take a few minutes, so I'll go ahead and jump to where the deployment has completed successfully.

Validating the Deployment

Okay, our deployment has completed successfully. We can see seven resources were successfully added. Let's go over to the AWS console and get the public IP address of our EC2 instance and validate that the web page is available. Okay, here we are in the EC2 console. I'll go ahead and refresh our view of instances. There is our instance that has been created. I can click on that and see that it does have a public DNS. So we can go ahead and grab that address, and I'll open up a new browser tab, and we can go to that address. And there you go. It has successfully loaded our web page. Congratulations! You did it. Have a taco. Going back to Visual Studio Code, since this was simply a demonstration environment, the last thing we can do is destroy the environment so it doesn't cost us any money, and we'll do that by running terraform destroy. Once you run terraform destroy, it will plan out the changes it needs to make to destroy everything that's in your environment, and then it will ask for confirmation. In the read out, the red dash indicates that something is going to be destroyed or removed, and now it's asking if we're sure we really want to do this. And we do, so I will type in yes, and now it will go through the process of removing all of those resources, and we no longer have to worry about paying for them. I encourage you to do this when you finish any exercise and you know you won't be coming back to the environment for a while. It's very simple to stand it back up by simply running terraform plan and apply again when you're ready to work with the environment.

Summary

Alright, let's sum up what we've learned in this module. Terraform is a tool used to automate infrastructure, which is way more fun than manually deploying stuff. Terraform itself is a single binary available for just about any operating system out there. The configurations Terraform uses are written in either HCL or JSON, although HCL is way more popular. And finally, the basic workflow for Terraform is initialization, plan, and then apply. The base configuration we just deployed is pretty simple and it leaves lots of room for improvement. In the next module, we're going to take a look at how we can use variables and outputs to improve our configuration. I'll see you there.

Using Input Variables and Outputs

Overview

All programming languages have a way to submit information into the software and retrieve output. Terraform is no different. In this module, we are going to explore how to use input

variables, local values, and outputs to improve our Terraform code making it more dynamic and reusable. Everyone this is Ned Bellavance. I'm a HashiCorp ambassador and founder of Ned in the Cloud. Let's dig into using input variables and outputs. The base configuration we deployed to AWS had all of its values hardcoded and provided us with no output. It's time to change that. We'll first start with learning how to supply input values to Terraform for use in a configuration, and then we'll learn how we can construct internal values inside the configuration for reuse. We'd also like to get some information out of our configuration once it's deployed and that is done through outputs. Finally, we are going to make a bunch of changes to our configuration, but what if we get something wrong? It sure would be nice to validate our config before we try and deploy it, and we'll see how Terraform has some built-in tools to help.

Working with Data in Terraform

Terraform can accept values as input, transform values inside a configuration, and return values as output. With that context, let's explore how to work with data inside Terraform. There are three different concepts to consider when working with data in a Terraform configuration. The first is called input variables, or just variables for short. Input variables are used to pass information to a Terraform configuration. The variables are defined inside the configuration, and the values are supplied when Terraform is executed. Local values, sometimes just called locals, are computed values inside the configuration that can be referenced throughout the config. In other programming languages, these would usually be called variables. The values for locals are not submitted directly from an external input, but they can be computed based on input variables and internal references. Data is returned by Terraform with output values. The outputs are defined in the configuration, and the value of each output will depend on what it references inside the configuration. Just like locals, the output value can be constructed from one or more elements. Since everything starts with inputs, let's take a closer look at input variables.

Input Variable Syntax

Variables are defined inside of a block just like everything else in Terraform. A variable block starts with the variable keyword followed by a single label, that is the name label. All the other properties of the variable are defined inside the block and all of those properties are optional. You can have a variable with no arguments and that's acceptable, although it's not really preferred. Let's take a look at the optional arguments inside the variable block. The type argument defines the data type associated with your variable and it provides a certain level of error checking. If you say the variables should be a number and someone submits a string, Terraform will throw an error. Now you might be wondering what data types are available to me. Don't worry, we'll cover that in the next section. The description argument helps provide some context for the user when they get an error and it will also be useful when we package configurations up in modules, but we'll cover that later in the course. The default argument allows you to set a default value for the variable. If no value is submitted for the variable, Terraform will use this default value. If you don't set a default value and none is submitted when the configuration is invoked, Terraform will prompt you at the command line to supply a value. The last argument I will cover is the sensitive argument. It accepts a

Boolean value of true or false. If it's set to true, Terraform will not show the value of the variable in its logs or the terminal output. This argument is useful when you have to submit potentially sensitive values like a password or an API key and you don't want them showing up in clear text in your logs or terminal output. Let's take a look at a few examples of actual variables and how to refer to their value inside a configuration. The first example shows a variable with the name label, billing_tag. No arguments are provided and none are needed. This is a quick and dirty way of adding a variable to a configuration. Since no default value is specified, you'll need to provide one at execution time. Our second variable has the name label aws_region, and this time, we have some arguments. We're going to set the type to string since the value will be one of the AWS regions and those are strings. We've got a helpful description here and we're setting a default value of us-east-1. So if no value is specified at execution time, Terraform will use us-east-1. Finally. this is not a sensitive value so we've set it to false. We don't actually have to set sensitive to false as it is false by default. To refer to the value stored in the variable, we simply use the var identifier dot the name_label. For instance, to refer to the value stored in our aws_region variable, the syntax would be var.aws_region and you would get back the string stored in the variable. Speaking of strings and data types, let's talk about the different data types that exist in Terraform.

Terraform Data Types

We can group the data types supported by Terraform into three categories. The most basic are the primitive data types. These are string, number, and Boolean. A string is a sequence of Unicode characters, a number can be an integer or a decimal, and Boolean is either true or false. The next category is collection data types, and they represent a grouping of the primitive data types. A list is an ordered group of elements, a set is an an unordered group of elements, and a map is a group of key-value pairs. In each case, the values stored in any of these collection data types must be of the same data type. The last group is structural data types, and they're very similar to collection data types, except they allow you to mix the data types stored in each grouping. Aside from that difference, tuples are functionally equivalent to lists and objects are basically equivalent to maps. It's useful to be aware of structural data types, but chances are you're not going to use them for basic configurations. They're more of an advanced topic. Let's take a look at some examples of the collection data types to help clarify things. Here's a couple examples of lists. Notice that each element of the list is of the same data type, all numbers in the first list and all strings in the second. The third list mixes data types, which would be invalid for a list, but valid for a tuple. Our Map example has three key-value pairs. The keys are going to be strings, and the values must all be the same data type. In this case, they are all of type string. You can create more complex structures using the object data type, but as I said, that's really beyond the scope of this course. If you want to use a collection for a variable, how do you construct it, and how do you reference the values inside? Let's take a look. Let's say we'd like to have a variable with a list of AWS regions. The type argument takes the form of the collection type we'd like to use and what data type will be stored in it. In this case, we have a list collection type that will be storing string values. For our default value, we have provided a list of four regions, each as strings. Lists are an ordered data type. We can refer to an element by number, starting with 0. If we want the first element in our list, which is us-east-1, our syntax would be var.aws_regions and a 0 for the first element enclosed in square brackets. We can get the whole list by only specifying the name label and skipping the square brackets. What if we want a map holding AWS instance

sizes? The type argument is basically the same. We want to have a map with strings as the value held in the map. For the default, we can define the keys as small, medium, or large and associate an EC2 instance size with each key. If we want to refer to the value stored in one of those keys, there are actually two ways of doing so. The first is var.<name_label>.<key_name>. The second is var.<name_label>, followed by the key_name in quotes inside of square brackets. We can retrieve the value stored in the small key by writing var.aws_instance_sizes.small or var.aws_instance_sizes, then small in quotes and brackets. Armed with our new knowledge of using variables, let's check in with the folks at Globomantics and see how we can improve our base configuration.

Globomantics Configuration Updates

Sally Sue is excited that you got the environment up so quickly, but the folks over in ops have some requests about how the environment is deployed. Let's review the current architecture and the requests for improvement. The current deployment architecture is a single EC2 instance in a public subnet inside a VPC in the us-east-1 region of AWS. The ops team doesn't want you to change the architecture yet, but they do want you to make some code improvements. John is from the ops team, and he has a little experience with Terraform. He's come up with a list of possible improvements for your code. For starters, those AWS credentials can't live in the code file. It's just not safe to throw those things around. John would like you to find a better way, preferably a way that doesn't store the credentials in a file at all. Speaking of hard-coded values, John would like you to use variables wherever possible so the configuration can be more dynamic and possibly reusable. Globomantics is also instituting default tags for their AWS resources, and John would like an easy way to apply default tags to all the resources in the config without doing a lot of find and replace. Finally, it would be nice to know the public DNS hostname of the EC2 instance without having to go to the AWS console. You tell John, no problem. We'll start by adding some variables.

Adding Variables to the Configuration

Now it's time to start adding some variables to our configuration. Now before you get started, if you destroyed the environment from the previous module, go ahead and recreate it now because we're going to be making changes to the configuration and then seeing how those changes apply to what's been deployed already. With that in mind, let's take a look at our current configuration by opening up the main.tf file in globo_web_app. There we go. This is our current main.tf file. Now, we wanted to find some variables for this configuration and the first thing we can do is create a file called variables.tf in the same directory. Remember, Terraform will put together any .tf files it finds in the same directory. By keeping the variables in a separate file, we can easily look between the main.tf file and the variables.tf file as we add new variables. I'm going to go ahead and hide the file tree and we'll split the view between variables.tf and the main.tf file. Now we can add variables in the variables.tf file and make the changes in the main file. Let's first start by getting rid of those AWS access key and secret key values. We'll start by creating a new variable, and remember, this starts with the variable keyword. We'll give it the name label, aws_access_key. This is going to be of type string. We can add a description of aws_access_key. We're not going to set a default for this variable because the whole point is getting the access key out of the configuration, but

we should set one more argument in here and that's setting sensitive to true. After all, we don't want this access key to be exposed in the logs or in the terminal output. Alright, now that we have our first variable, let's go ahead and replace the hardcoded string with a reference to this variable. We'll do that by removing the current value and now we'll add a reference to our variable. Remember that goes var. the name of the variable, which is aws_access_key. If you're using VS Code or something that has similar plugins. it might even helpfully finish that for you. Now let's go ahead and do the same with the AWS secret key. So I'm going to copy this variable and paste it down below, and I'm going to change the name from access key to secret key, I'll change the description, and now we'll replace the secret key value with a reference to our variable. There we go. Now our access key and our secret key are no longer hardcoded into our configuration. Let's also take this opportunity to add a variable for our region in case we wanted to deploy to a different AWS region. We'll start with the variable keyword and we'll set this variable to aws_region. Just like the keys, this is going to be of type string. We'll set a description of AWS Region to use for resources, and let's set a default value for this variable of us-east-1. Now this is not a sensitive value, so we won't set the sensitive argument since it defaults to false. Let's go ahead and replace the region with our variable. We'll set it to var.aws_region. Alright, our provider is now using all variables for its values. That's great. Let's scroll down a little bit more and see where else we could use variables. In our networking configuration, we can see the CIDR block has a hardcoded value, enable DNS hostnames has a hardcoded value, and in the subnet, the CIDR block and the map_public_ip_on_launch both have hardcoded values. Scrolling down a bit more, in the security groups, you could potentially set variables for the port numbers if you would like to, and scrolling down beyond that, there is an instance type which is hardcoded for the AWS instance, that's another good place where we could add a variable. What I'd like you to do now is pause the video and try to add all these variables to your configuration. When you're done, you can go ahead and look in the file layout for the M for solution and that will show you the variables that I added to the configuration and how I reference them in the main.tf file. So go ahead and pause now, try to add the variables, and we'll resume in a moment. Alright, let's see how you did. Looking in the variables file, we've got our access key, secret key, and region that we created. Scrolling down some more, we've got enable_dns_hostnames, the vpc_cidr_block, the vpc_subnet1_cidr_block, and map_public_ip_on_launch. And scrolling down a bit more, we have the instance type and there should be references in the main.tf file for each of these variables. The next thing that we're going to talk about is local values and how we can use those to add those common tags that John was asking for.

Local Values Syntax

As I mentioned earlier, local values are values computed inside of the configuration. You can't submit values directly to them, unlike input variables. The syntax for locals starts with the keyword locals, and that's it for labels on the block. The rest of the information goes inside of the block. Inside the block are key value pairs. The value can be any supported Terraform data type, string, list, object, the sky is the limit, or more accurately, the supported data types are the limit. Here's an example of a locals block. The first key value pair defines a local with the name instance_prefix and the value globo. The next key is common_tags, and its value is a map defining some common tags. You can refer to other values inside of your configuration for the values on local. For instance, the project key is being assigned the value

in the variable project. Hey, this seems pretty useful for our Globomantics requirements. You can specify the locals block multiple times in your configuration if you want to, but the name of each key must be unique within the configuration since that is how you reference a locals value. To refer to the value stored in a local, the syntax starts with the local keyword. Note that local is singular, not plural, that kind of threw me off the first time I saw it, followed by dot and the name_label. To get the value in the instance prefix, the syntax would be local.instance_prefix. If we've got a collection data type in our local, the same syntax we saw from the variable example applies. We could get the value stored in company by writing local.common_tags.company. If we'd rather get the entire map, we only need to write local.common_tags. Let's head back to our configuration and update it based on this new information.

Adding Locals to the Configuration

Globomantics is looking to add three common tags to start to all resources we've defined in our configuration. We can define the common tags in a locals value and then use that value throughout our configuration. Let's start by creating a new file called locals.tf. All right, and in our locals.tf, we will start by defining a locals block. Within our locals block, let's go ahead and define a map of common tags. We'll start with common_tags =, and then we'll use the curly braces to specify a map data type, and then we'll add our key-value pairs. The three values they want to start are company, project, and billing_code. Now where are we going to get this information from? Let's use variables to get this information. Let's open up our variables file and add three variables for company, project, and billing_code. We'll scroll down to the bottom of our variables file, and we'll go ahead and add those three variables. Now here's a chance for you to take the wheel again. Why don't you pause the video and add those three variables. Again, they are company, project, and billing_code. All right, I have added those variables to my configuration, and as you can see, I set a default of Globomantics for the company and specified no default value for the project or the billing_code. Now let's go ahead and add these variable values to our locals. And we can do the same thing we did before, which is hide the File Explorer. We'll split variables out to the right side so it's easier to work with and showed the locals on the left side so we remember exactly what we're working with. We'll start by adding company. Next, we'll add project. And for this one, John has requested that the project be the company name dash the project name for the value. Now how do we go about referencing a variable inside a larger string? We're going to use interpolation syntax, which sounds real fancy, but it's actually quite easy. We start by adding quotes to indicate that this is going to be a string, and then we need to reference our variable. We start with a dollar sign followed by curly braces. This let's Terraform know that we're going to be referencing a value from a variable or some other object within our configuration. Now we can add the variable reference, which will be var.company. Next, we're going to add a dash after the curly braces and another reference to the project value stored in the variable project. Now we've created a string from our two variables that is of the form company-project. Lastly, let's add our billing_code. And there we go. We have successfully created our common_tags local value. The next thing to do is add this common_tags value to our main.tf file for each AWS resource that supports tags. Let's go ahead and add the first one together. So I'll switch over to the main.tf file, and let's go down to our first resource, which is the aws_vpc. Within the configuration block, I'll go ahead and add tags, and I'm going to set tags equal to local.common_tags. Now this map will be

submitted to the tags argument, which expects a data type of map, and those tags will be applied to the VPC. My challenge to you now is to pause the video and add this tags argument to all the other resources in our configuration that support a tags argument. The only resource that does not is the aws_route_table_association, so you can add the tags argument to all the other resources in the configuration. All right, I have successfully added the tags argument to all of my resources, and hopefully you have too. The last thing we're going to do is add an output so that we know what the public DNS hostname is for our EC2 instance. So let's talk about outputs now.

Output Values Syntax

Output values are how we get information out of Terraform. Outputs are printed out to the terminal window at the end of a configuration run. It also exposes values when a configuration is placed inside a module, something we'll cover later in the course. The syntax for an output starts with the output keyword followed by the name_label for the output. Inside the configuration block, the only required argument is the value of the output. Just like the value of a local, the value of an output can be any supported Terraform data type. You can return a simple string or a complex object. Optional arguments include the description, which is only seen when looking at the code for a configuration, so it's not all that useful. The sensitive argument will set an output to sensitive, meaning that the actual value will not be printed in the terminal. This is useful when you want to pass a value from one module to another and avoid having it printed in clear text in the logs or the terminal. Trust me, that will make more sense when we get to modules. Here's our example of an output with the name_label public_dns_hostname. We're setting the value equal to the public_dns attribute of our web_server EC2 instance using the same reference syntax we saw in the previous module. We've included a description of the output, too, for our own personal reference. Sensitive is not set, so it defaults to a value of false. That's good, because we want this value printed to the terminal window so we can use it. Let's head back to the configuration and add an output.

Adding Outputs to the Configuration

Just as we did with the locals and with the variables, let's go ahead and add a file for the outputs. Within the outputs, we are going to define a single output. We'll start with the output keyword, and we'll set the name to aws_instance_public_dns. For the value, we're going to reference our EC2 instance. So let's go ahead and go back into split-screen mode here and bring up our main.tf file. We'll scroll down to our instance definition, and for that we'll go ahead and copy the type, paste it in, we'll add the name_label, nginx1, and we'll add the property of public_dns. You can see that because we've initialized our configuration previously and I have the Terraform extension installed in VS Code, it knows all the attributes that are available for the aws_instance resource type, and so I don't have to remember all of them or look them up. I'll go ahead and save the file, and now we've added our single output that we wanted. Now you might be wondering, how do I know that I got all of this configuration syntax correct? Well, your code editor should help you a little bit by highlighting improper syntax, but Terraform can also help you with the validate command. Let's learn more about that now.

Validate the Configuration

Before we try to apply our update, it would be nice to know if the configuration is syntactically correct. Our linter does its best to help, but Terraform can also lend a helping hand. Terraform has a command called validate that will help you make sure your configuration is correct. Before you run the command, you'll need to run Terraform init. That's because it's checking the syntax and arguments of the resources in the providers, and it needs the provider plugins to do so. When you run validate, it will check the syntax and the logic of your configuration to make sure everything looks good. If it finds any errors, it will print out the error and the line where it found the issue. Sometimes it will even make a suggestion. Terraform validate does not check the current state of your deployment; it's just verifying the contents of your configuration. It also carries no guarantee that your updated deployment will be successful. Your syntax and logic might be correct, but the deployment could still fail for any number of reasons; insufficient capacity, incorrect instance size, overlapping address space. Validate does what it can, but it can't do everything. Why don't we try to use validate against our updated configuration?

Using the Validate Command

The configuration we have now should pass validation, but we want to see what validation does, so let's go ahead and add a couple errors to our main.tf file and then see how Terraform catches them. Let's start by scrolling down, and we'll put some square brackets around enable_dns_hostnames as if it were a list and not a Boolean value. That should definitely throw an error. We can also use a variable reference that does not exist, so let's go ahead and delete block off var.vpc_cidr, and that should also throw an error. Now that we've made those changes, we'll go ahead and save the file and we'll bring up the terminal to run terraform validate. I'm already in the globo_web_app working directory, and I've run terraform init, so all I should need to do is run terraform validate. And as you can see, Terraform has come back with an error. Let's go ahead and expand the window so we can see the full error. Here it's telling us that we have a reference to an undeclared input variable, vpc_cidr. Okay, well, we knew that, so let's go ahead and fix that problem. We'll scroll down here and we'll add _block back to our variable name label. There we go. I'll go ahead and save the file, and now we can run terraform validate again. Now we can see that Terraform says we have an incorrect attribute value type. The value attribute for enable_dns_hostnames should be a Boolean, and we've supplied a list, so let's go ahead and take those square brackets off and save our file once more, and we'll go ahead and run terraform validate a third time. Success. Our configuration is valid. Now you may find that you have other issues in your configuration if you've been working on your own, so go ahead and remediate those now. The next step in our process is to supply values for the variables we've defined in our configuration. But how do we go about doing that? Let's find out.

Supplying Variable Values

When it comes to setting the value for a variable, there are at least six ways of doing so. That's a lot! The easiest way to set a value is to set the value with the default argument. We've already seen that in our configuration. You can also set the variable at the command line when executing a terraform run. You can use the -var flag followed by the

name of the variable and the value you want to set it equal to. You can repeat this flag for each variable you'd like to set. You can also have all your variable values in a file and submit that file with the -var-file argument. Inside the file will be each variable name label as a key, followed by an equal sign and the value for the variable. There are two other ways to submit values from a file. If there is a file in the same directory as the configuration named terraform.tfvars or terraform.tfvars.json, which needs to be properly formatted JSON, Terraform will use the values it finds in that file. Additionally, if there is a file in the same directory as the configuration ending in .auto.tfvars or .auto.tfvars.json, Terraform will use those values as well. The final option is to use environment variables. Terraform will look for any environment variables that start with TF_VAR_ followed by the variable name. If you don't submit a value for a variable in any of these ways, Terraform will prompt you for a value at runtime. I know that's a lot of options. And what if you set the same variable in multiple ways, what's going to happen? There is an order of precedence. Here's what it looks like, but I have to admit, I always have to look it up. Terraform evaluates each of these options from left to right, with the last one winning. If you find that your variable has the wrong value being set, this might be the culprit. Now that we know how to set values for our variables, let's go make use of our updated and validated deployment.

Deploying the Updated Configuration

Reviewing the variables in our configuration, there are a few that don't have a default set. We're going to have to set the aws_access_key and aws_secret_key. Scrolling down to the bottom of our variables, we can see that the project and the billing code also need a value supplied. All the other variables have a default set and we can go ahead and keep using that default. Let's take a look at the potential syntax if we wanted to submit values for all of these variables at the command line. I'll go ahead and expand commands over here and we'll take a look in m4 commands. In our m4 commands, we have already initialized and validated our Terraform configuration. Now we can pass our variables at the command line if we'd like to and the syntax for that is -var= the name label of the variable and then another equals and the value you want to set that variable to. Now, as you can see, this command can get very long. There has got to be a simpler way to do this and we know there is. Let's go ahead and create a file called terraform.tfvars and populate it with some of our non-sensitive variables and values. I'll go ahead and create a new file called terraform.tfvars in the same directory as our configuration. There we go. And let's go ahead and split screen things again so we can add the values and see what the values should be. We'll start with the first variable, billing_code. I'll go ahead and grab that value and paste it over in our file and I'm going to set it equal to the value described in the command. There we go. Now let's set the next variable, which is project, go ahead and paste that over and set that one equal to web-app. There we go. And the next two variables are AWS access key and secret key. We don't want those in a file, instead, we can store them inside an environment variable. Let's go ahead and save our terraform.tfvars file, we'll go ahead and close that out, and then looking at our m4 commands, we can see we are going to export 2 environment variables. If you're working in Linux or Mac OS, you can use the export command. If you're using PowerShell, you can use the $env: and then the name of the environment variable. Once again, the environment variable is going to be TF_VAR_ the name of the variable that you want to set a value to. So the first one is going to be aws_access_key. I'm working in PowerShell so I'm going to go ahead and paste in my access key and secret key so I can set them as environment

variables. Alright, I've updated the two commands with my AWS access key and secret key. Let's go ahead and copy both those commands and paste them in the terminal down below. Now I have those environment variables set, they're not stored in a file with our configuration and they won't be shown in the terminal output or in any logging we enable for Terraform. Let's go ahead and clear the terminal and now we can go ahead and run Terraform plan with our output file and we don't have to worry about including any variable values in the command because we have defined them in files and environment variables. We'll go ahead and run Terraform plan now, and because the only change we made is to create variables, local values, and outputs, and add some tags to our resources, the only real change on the AWS side is to add those tags. So let's go ahead and expand the terminal up and take a look at what's changing in our configuration. Now we can see that there are six changes with nothing to add and nothing to destroy. If we look at what's being changed about our VPC, the yellow tilde means that something is being updated or changed, and the green plus sign means a value is being added so we can see the tags that are being added to our VPC. Let's go ahead and run the terraform_apply to update the tags on all of our resources. This should go very quickly because we're simply modifying the tags for our resources, it's not having to create or destroy anything. Now we can see that the output we get at the end is, in fact, the public DNS of our AWS EC2 instance. Now we can use that as opposed to going into the console. At this point, we have accomplished all the goals that John set out for us.

Summary

In this module, we made our configuration a bit more viable. We used input variables so we can supply the proper values at runtime and get those hard-coded credentials out of here. We also saw the multitude of ways to set values for our variables. It can get confusing quickly, so I recommend keeping it simple. We also managed to get some information out of our configuration with output values. And finally, we validated our configuration before trying to deploy it to catch any syntax or logic errors in our code. With a viable configuration in place, it's time to turn our attention back to the architecture. Our current design isn't exactly resilient, so we're going to add another instance and load balancing to the deployment. That's coming up in the next module.

Updating Your Configuration with More Resources

Overview

We've updated our configuration to include variables, locals, and outputs. Now it's time to update the architecture of our deployment to include resiliency by adding new resources. Hey, everyone. This is Ned Bellavance. I'm a HashiCorp ambassador and founder of Ned in the Cloud. Let's add some more resources to our configuration. Our existing architecture is a single EC2 instance running in a single subnet on AWS. If something were to happen to that instance or the availability zone associated with the subnet, our application would go down. That's probably okay for development, but not if this application is going to make its way to production. We will start our process by updating the architecture

design, determining what new resources we need to add. Once we know what resources we need to add, we can consult the official HashiCorp docs to see what arguments are required for each resource. Armed with the knowledge gleaned from the docs, we will set about updating the configuration with new resources and data sources and apply the updated config to our existing deployment. We will also take a moment to talk a bit more about the magical state data Terraform uses to map a config to a deployment. What's in that data, and how should you interact with it? First, we will start by planning and infrastructure update with our friend John.

Globomantics Architecture Updates

Adding variables, locals, and outputs to the configuration was a great start, but now it's time to add some resources to improve the architecture. Let's see what Globomantics has in mind. Our current architecture is using a single subnet in a single availability zone with a single EC2 instance. That's a lot of single points of failure. John from the Ops Team has some suggestions to improve the reliability of the deployment. First, we'll start by adding a second availability zone in AWS. If you're not familiar, each availability zone in an AWS region is a separate physical data center, and each subnet is associated with one, and only one, availability zone. Adding a second subnet in a separate availability zone will protect from a zone failure. We also only have a single EC2 instance. John suggests that we add a second instance in case the first instance fails. Of course, adding a second instance doesn't magically fix things; we need a way to make both instances accessible, and we'll do that through a load balancer. Lastly, John wants to make sure we maintain the readability of the code. He noted that we made a separate file for variables, locals, and outputs, and he thinks it would be a good idea to split the resources out as well. Perhaps we could make one for base networking, another for instances, and one for the load balancer. Not only does that make it easier to read the code, it might make some files reusable in other configurations. What does this updated architecture look like? Here's our current architecture with the single subnet and EC2 instance. We didn't specify an availability zone for our subnets, so AWS picked one for us. And here's the new architecture. We now have two subnets that should be in separate availability zones, meaning we're going to need to specify an availability zone for each one. We now have two instances that will be identical in nature except for the subnet they attach to, and we are adding an application load balancer, which will serve as the public endpoint for our application and direct traffic to our instances. If you're not overly familiar with AWS, or even if you are, you might be wondering exactly which resources you'll need to add to the Terraform configuration to create this architecture. Well, I'm not going to leave you hanging to figure that out on your own. This is a Terraform course, after all, and we're not here to learn the intricacies of AWS. Here are the new data sources and resources we'll need to add to our configuration. With our two subnets, we now care which availability zone each one is in. We could add a variable to specify the availability zone for each subnet, but there's a better and more dynamic way. We can add a data source that gets the list of availability zones in the current region and use that list in our subnet settings. For the load balancer component, there are actually several resources that need to be added, and I have to admit, it's not immediately obvious what they are. The first is the aws_lb resource itself, which will be the application load balancer. Next will be the aws_lb_target_group, which defines a group that the application load balancer can target when a request comes in. To service incoming requests, we need an aws_lb_listener that listens on port 80 for inbound requests.

And lastly, we need to associate our target group with our EC2 instances. The aws_lb_target_group_attachment resource takes care of that. Wow, that's a mouthful, isn't it? Well, that's all the new resource types. We're also going to add an additional subnet, EC2 instance, and a security group for the load balancer. Why don't we head over to the configuration and add some placeholders for the new resources?

Adding New Resources to the Configuration

All right, let's get started by updating the file structure a little bit for our globo_web_app. I'll go ahead and expand the folder out now, and we'll start by creating some new files. Let's go ahead and create one for the load balancer, and we will create one for the instances, and we can rename our main.tf network because the only thing that's going to be left in it once we move stuff around is networking components. Okay, now that we've created our files, let's go ahead and move the instance configuration into its own file. Scrolling down to the bottom, I'll go ahead and grab this entire body of text that defines the instance, cut it, go into the instance file, and paste it in there. All right, our instances will now have their own file to reside in. I'll go ahead and save that. For the load balancer, we haven't actually created any of the resources yet, so let's instead add some placeholders for the resources we know we need to create. I often add comments that let me know what resources I need to create before I actually create them, so I'll go ahead and add some comments to this file now. I'll add the aws_lb, the aws_lb_target_group, the aws_lb_listener, and the aws_lb_target_group_attachment. Now I know what needs to go in this file, but, of course, I still have to create the resources and understand all the arguments that go into each resource. How am I going to get that information? Ah, the answer is to read the documentation.

Using the Documentation

This should go without saying, but I'm going to say it anyway, there is no shame in going to the docs to try and figure out how to configure a resource or work with some Terraform syntax. Whenever I am writing a new Terraform configuration, I usually have the code editor open in one monitor and multiple docs tabs open in another monitor. So why don't we go check out those docs? The documentation for the providers and the resources within them can be found at registry.terraform.io. Now, the provider that we are interested in is the AWS provider which we can find by simply clicking on Browse Providers and it gives us a list of the most popular providers on the first page. We want to work with the AWS provider, and lucky us, it's right there. Let's go ahead and click on that. The front page will tell you some information about the current AWS provider and you'll note there is a tab for documentation. Let's go ahead and click on that to go to the documentation. The main page of the documentation explains a little bit about how to use the AWS provider. It gives some examples of how to instantiate it, as well as how to authenticate to the provider. We'll cover that more in a leader module. Right now, we are mostly concerned with adding our data source and our resources. The easiest way to find those is usually to search through the filter box so I'll go ahead and start typing in availability zones. After typing just a portion of the phrase, we can see under Data Sources in the matching results, we have aws_availability_zone and aws_availability_zones, that's the one I'm interested in so I'll

go ahead and click on it, and this is the documentation for the data source. It gives us a description of what the data source does and then it provides us a simple example for usage. This is great. This explains how to use this data source. First, we declare it using the type of data source and giving it a name label, and then there is some optional arguments that go in the configuration block. When we want to use this data source. It gives us an example of how to use it with a subnet, which is pretty convenient because that's exactly how we want to use this data source. If we want to inspect some more information about this data source, we can go to the arguments reference. I'll go ahead and click on the link. This takes us down to the Arguments Reference portion of the page. These are the arguments you can supply inside of the configuration block. We might want to specify the state argument that filters the list of availability zones that are returned by the data source. We could say just give me the available ones. Below the argument reference is the attribute reference. These are the attributes that are exposed for the data source. The one that we're most interested in is the names of the availability zones because that's what we're going to use to configure our subnet, and based off the information here, it is a list that is returned of the availability zone names, which means we can reference each zone by its element within the list. If we go back up to the example usage, that makes sense with what it's showing us here that we use the names attribute along with an element of that list to retrieve a single availability zone name. Now that we know this, we can go ahead and just copy a portion of this example and put it right into our configuration file. There is no need to recreate the wheel here. With that text copied, let's go over to our configuration, and since this is going to be part of our network configuration, let's go to the network file, we'll scroll all the way up to the top, and we'll add another data source in the data area. There we go. We've added our new data source for the availability zones. Now we can make use of them in our subnet. The next thing to do is add the additional subnet, security group, and EC2 instance.

Updating the Network and Instance Configuration

Now that we've successfully added our availability zone's data source, it's time to update our subnet's security group and EC2 instances. Let's first start by updating our existing subnet to use an availability zone. I'll go ahead and scroll down in the file to our subnet. There it is. And we're going to add a new argument here for the availability zone. The argument is going to be availability zone, and I will set that equal to the data.aws_availability_zones.available data source. And the attribute we want, remember, is names. So we'll do .names. And for this first subnet, let's take the first element from the list. So we'll do square brackets and 0, since lists are 0-indexed. All right, we've added our availability zone. Now before we create a second subnet, the CIDR block is defined with the variable vpc_subnet1_cidr_block. Wouldn't it be easier if we had a variable that had all of the subnet blocks defined in it? Why don't we go ahead and create that first. So, we'll go ahead and open up variables, and we'll scroll up to that variable definition for the subnet1_cidr_block, and instead we'll change that to vpc_subnets_cidr_block. Instead of type string, we'll make it a list of strings. And for the default, we can update this to a list of strings. I'll add the square bracket, so it's a list, and I will add a second element to the list of 10.0.1.0/24 for our second subnet and close the square bracket, and we can update our description to CIDR Block for Subnets in VPC. All right, let's go back to our subnet configuration in the network.tf file. And we'll update this variable to vpc_subnets_cidr_block, and we'll select the first element out of the list. Now you'll see the reason I did this in a moment as we add the second subnet for our

configuration. Before I add the second subnet to our configuration, my challenge to you is to go ahead and try to add these resources on your own. You're going to need to add a second subnet, a route table association, and a second EC2 instance in the instances file. Go ahead and try to do that now. If you run into trouble, you can always check the solution that's in the m5_solution directory, and we'll come back in a moment to see my updated solution. All right, we're back. Let's go ahead and see what I did in my solution. For the second subnet, I made a copy of the existing subnet resource, and I changed the name label to subnet2. For the cidr_block, I changed the element to 2 to reference the second element in the list, and for availability_zones, I changed that to a 1 as well to reference the second availability zone. By setting it up in this way, we could add a third or fourth subnet and just make sure that we update the subnet's cidr_block appropriately. Scrolling down into the routing, let's take a look at those route table associations. Once again, I simply copied the existing route_table_association_resource, changed the name label to subnet2, and changed the subnet_id reference to subnet2. Both of these subnets are going to use the same route table. Moving over to the instances file, for the second instance, once again, I made a copy of the existing aws_instance resource, I changed the name label to nginx2, and I changed the subnet to subnet2. To differentiate the two web pages, I changed the echo command for the first one to say Taco Team Server 1, and if we scroll down, the second one is now Taco Team Server 2. The next thing we need to do is create an additional security group for our load balancer so it allows port 80 traffic from anywhere. So let's go back to the network file. And if we scroll down here, we already have a security group for our instances that allows HTTP access from anywhere. Let's make a copy of the security group. I'll go ahead and update the name label of this new security group. We'll call it alb_sg, and we'll update the name to nginx_alb_sg. We're gonna be using the same VPC ID, and the ingress block is already correct. We want to allow port 80 access from anywhere. We do want to make a change to our existing security group for the instances. Now that we have a load balancer in front of them, they should only accept traffic from addresses that are within the VPC. So we can scroll up and change this cidr_block reference to var.vpc_cidr_block. Now, it will only allow traffic from addresses that are in the vpc_cidr_block. Let's go ahead and save the network file. The next thing to do is add our load balancer resources. But before we do that, we need to know how to construct each of those resources. Let's head back to the docs.

Adding the Load Balancer Resources

Back in the docs, let's try to search for aws_lb, and wow, that returns over 1000 results. Okay, that's not going to be very helpful. Let's go ahead and clear the filter, and I happen to know from experience that this is under Elastic Load Balancing v2. So let's scroll down to that. There's Elastic Load Balancing v2. That includes the application load bouncer and the network load balancer. We'll go ahead and expand that out, and we can see it split up into Resources and Data Sources. That means if we had an existing AWS load bouncer and we wanted to use it as a data source, we could. But in our case we want to create an AWS load balancer resource, so let's go ahead and click on that resource. And just like the data source, this gives us an example usage for both the application load bouncer and the network load balancer. The example here is very close to what we actually want, so let's go ahead and copy this and place it in our configuration and then make a few simple updates. I'll go ahead and copy the text and go over to the configuration, and we'll paste it directly in the file. There we go. Now let's update the name label and the name of our load balancer. We'll set the name

label to nginx, and we'll set the name to globo_web_alb. Internal should be set to false because we're creating a public load balancer. The load_balancer_type should be application. That's the type that we want. For the list of security_groups, we should update it to the security group that we just created. So let's go into split screen mode and bring up our network configuration on the left and scroll down to our new load balancer security group and grab that name label, and we'll paste it over here. And now we need to update the subnets argument to the list of subnets that we're going to be using. So we'll go ahead and delete this subnets argument and add square brackets to indicate a list, and now we can add our two subnets. So I'll go ahead and scroll up to our subnet definitions, and I'll grab the resource type, followed by the name label, and the attribute that we want is id, so I'll do .id, then I'll add a comma, and we'll copy this text, paste it, and update it to subnet2. Now we've included both of the subnets we want to associate with our load balancer. The next property is enable_deletion_protection. We're going to set that to false because we want Terraform to be able to delete this load balancer when we're done with it. For now, we're not going to configure the access logs, so I'll go ahead and delete this block. And lastly, we'll update our tags to reference our local value, local.common_tags. That's everything for the load balancer. My challenge to you now is to create the rest of the resources using the documentation. If you get stuck, you can reference the solution that's in the m5_solution directory, and when we come back, you can see how I updated my solution. Okay, here's my updated solution, and why don't I get out of split screen mode here so we can better see what's in the configuration. I added the aws_lb_target_group, specifying the correct port of 80, the protocol of http, and the correct VPC ID. Scrolling down a bit more, I created the aws_lb_listener, which needs to reference the ARN of the load balancer that we've created, the proper port and protocol, and we're going to set the default_action in here of type forward to send traffic to a target group, and then we'll specify the target_group_arn of the target group we just created. Scrolling down a little bit more, we have two target group attachments, one for each EC2 instance. In there I've specified the target_group_arn, the target_id is going to be the idea of each EC2 instance, and the ports is going to be ports 80. That's everything that goes into this configuration. Now before we apply our updated configuration to our deployment, let's talk a little bit about Terraform state.

Viewing State Data

So far, we've talked about state data as the way that Terraform maps what's in your configuration to the actual deployment on target environment, but what's in that state data and how can you interact with it when necessary? Terraform state data is stored in a JSON format. You should not try to alter this JSON data by hand. Bad things can and will happen. We'll be looking at how you can use Terraform commands to work with it shortly. State data stores important information about your deployment, including mappings of resources from the identifier in the configuration to a unique identifier in your target environment. Each time Terraform performs an operation like a plan or apply, it refreshes the state data by querying the deployment environment. The state data also contains metadata about the version of Terraform used, the version of the state data format, and the serial number of the current state data. When Terraform is executing an operation that potentially alters state data, it tries to place a lock on the data so no other instance of Terraform can make changes. Imagine if the state data was in a shared location and two admins tried to make conflicting changes at the same time, that's no good. Locking helps to prevent that

situation from arising. Speaking of the state data location, you can store the state locally on your file system, which is what Terraform does by default, or you can specify a remote backend for the state data, that could be an AWS S3 bucket, an Azure storage account, an NFS share, or HashiCorp's Terraform Cloud service. A remote backend for state is useful when working on a team to collaborate and to move state data off your local machine for safety's sake. We're not going to cover remote state data in this course, but if it sounds interesting to you, I recommend checking out my Terraform deep dive course to learn more. Another feature supported by Terraform state is workspaces which enable you to use the same configuration to spin up multiple instances of a deployment, each with their own separate state data. We will cover workspaces in more detail in a future module. What does state data look like? Here is a rough sketch of what is in the state data. We've got the current version of the state data format and the version of Terraform that was last used on the data. This is important because older versions of Terraform might not be compatible with the latest format of the state data. Terraform will let you know if that's a problem. The serial number is incremented each time the state data is updated. The lineage is a unique ID associated with each instance of state data and prevents Terraform from updating the wrong state data associated with a config. The Output section contains the outputs we saw printed in the terminal window in the last module, and resources is a list of resource mapping and attributes. Let's jump over to our configuration and look at the actual state file. Back in our configuration, we can see the state file is terraform.tfstate. Let's go ahead and open it. Starting from the top, the version is version 4, the terraform version used is 1.0.8, and the serial number is 32, that is incremented each time the state is changed. And then we have our lineage, which is the unique ID for this particular state data. Below that, we have the outputs. We have a single output defined and the value is stored in the state. Below that, we have a list of resources. The data source is considered a resource, in this case, and it has information about that data source. If we scroll down some more, we have our first actual resource, which is our AWS instance. It has the name label we've associated with this specific resource, the provider that was used to create it, and then information about that resource including its attributes. That is what you'll find if you look inside the state file, which leads me to another very important point. You do not want to make any changes to this file directly and honestly you probably shouldn't even open the file, in general. Let's look at some commands you can use to work with state data.

Terraform State Commands

There are a subset of commands with Terraform specifically to deal with state. We won't cover all of the commands, but I did want to touch on some of the most commonly used ones. To see all the resources being managed by Terraform, you can run terraform state list. From that list, you might want to know more about a specific resource. You can find out more by running terraform state show and the resource address, which is the resource type and the name label. You can move an item to a different address in the same state file. This can be useful for renaming resources or moving them into modules. The syntax for that command is terraform state mv for move, followed by the source address and the destination address for the resource. Lastly, if you need to purge something from the state, you can do so by using terraform st rm and the address of the resource. You might want to remove a resource from Terraform management without destroying it. You could remove the resource block from the configuration and then remove the entry from state. Otherwise, the next time you ran

terraform apply, it would attempt to destroy the deployed resource in the target environment, which leads me to my next and maybe most important point. The first rule of Terraform is to make all changes with Terraform. Don't try to manually edit state data, and don't make changes to managed resources with the cloud console or the CLI. Make changes in the configuration, and then apply those changes through Terraform; otherwise, Terraform will either undo your changes at best or get hopelessly confused at worst. With that advice in mind, let's head back to our configuration and get our updates deployed. Back in our configuration, let's go ahead and expand the commands directory and open up the m5_commands. First, let's try out a couple of those Terraform state commands. I'll go ahead and bring up the terminal down below, and let's first run terraform state list. I'll go ahead and expand the window a little bit here and here are all of the resources it knows about in state. If we want to see the properties of a specific resource, we can use the address that's shown on the screen. Let's run terraform state show and then look at the information for the aws_instance.nginx1. Okay, this now shows us all of the available information about nginx1. I'll go ahead and maximize the terminal, and we can see all of the information that's in here, and it's a fairly significant amount. Next up, let's validate our configuration and run the update against our existing deployment.

Deploying the Updated Architecture

Before we try to run a plan for our configuration, let's first run terraform validate and make sure we don't have any mistakes in our configuration. And as you can see, it caught an error in the loadbalancer.tf file. It's letting us know that we can't use underscores in the name for our load balancer. Okay, let's go into loadbalancer and update the name with dashes instead. And I'll go ahead and save the file and we'll run terraform validate again. Excellent! Now our configuration is valid. Let's go ahead and run terraform plan. I'll go back to my m5_commands. If you haven't already, you're going to need to export the environment variable TF_VAR_aws_access_key and secret_key. I've already done that, so I can scroll down to the terraform plan command. I'll go ahead and run that now, and we'll save the plan to m5.tfplan. Okay, we're going to be making some significant changes here. We have 12 things to add, 1 to change, and 3 to destroy. I'll go ahead and expand the terminal so we can see what is going on in the plan. Scrolling up a bit, let's see what's being replaced. Well, subnet1 needs to be replaced because it's changing availability zones, so it's letting us know it's going to delete that subnet and recreate it. Scrolling up a bit more, our security group for the NGINX instances is going to be updated because we're changing the ingress block. Our route table association for subnet1 has to be replaced because the subnet is being replaced. And scrolling all the way up from there to our first nginx1 instance, that also has to be replaced in part because we're changing the subnet ID, but also because we changed the user data that's associated with the instance. Now we are fine with all of these changes, so let's go ahead and run terraform apply to apply the changes to our target environment. This is going to take a little while because it's going to create that subnet and the load balancer and the EC2 instance. Generally speaking, the load balancer is what actually takes the longest to create, so I'll go ahead and pause the recording now and we'll jump to when the deployment has completed successfully. Our deployment is successful, but our output is still giving us the AWS instance public DNS. That's something we should probably change. Let's go ahead and open up outputs, and instead of the instance, we want to get the public DNS of our load balancer. Let's go back in split screen mode and we'll hide the terminal for a moment, and

let's go to the load balancer. We'll grab the address for the load balancer, which is aws_lb.nginx.dns_name for the attribute. And we'll go ahead and save that output. We'll bring the terminal back up. We can run a terraform validate to make sure our change didn't mess anything up. And now, because we haven't made any changes to the resources, we can just run terraform apply directly, and we can do that by simply doing terraform apply and adding the flag -auto-approve so it doesn't prompt us to approve any changes. I would only recommend doing this when you're absolutely certain that no changes will be made that might be destructive to your target environment. Since we're only changing an output, it's not going to make any changes to our target environment, and now we have the public DNS for our load balancer. Let's go ahead and grab that load balancer address and plug it into a browser. I'll go ahead and open up a new tab here and paste this into the browser. And if we look at the tab, it says Taco Team Server 2. If we refresh, it now says Taco Team Server 1. We know that both instances are responding on the load balancer and our deployment is successful.

Summary

In this module, we've added new resources to our configuration to make it more resilient and production ready. We also took a look at the docs for the AWS provider to help us with the arguments and syntax for all our new resources. Just remember, there is no shame in reading the docs or copying examples. State data is Terraform's map from the config to the deployment and it is very important. A corrupt state is a dire circumstance to find yourself in. Treat the state data with respect and all will be well. So far, we've only worked with the AWS provider. Now it's time to see how you add an additional provider, how to work with provider versions, and we're going to learn what provisioners are and why you probably shouldn't use them. That's all coming up in the next module.

Adding a New Provider to Your Configuration

Overview

One of the key strengths of Terraform is its vendor agnostic and pluggable approach. Anyone can develop a provider plugin for Terraform and you can use more than one provider in a configuration. We will see how to add and configure providers in this module. Hey everyone. This is Ned Bellavance. I'm a HashiCorp ambassador and founder of Ned in the Cloud. Let's add a new provider to our configuration. Our configuration will continue to evolve in this module based on requests from both the development and ops teams at Globomantics. One of their requests will require adding a new provider to the configuration, but before we do that, we'll learn a bit more about how to add and configure a provider. We are also going to dig into the dependency graph that Terraform creates when planning a deployment and learn when it might be necessary to specify an explicit dependency in your configuration. Finally, we are going to examine the options that exist for post deployment configuration of resources. Once our EC2 instances are deployed, how do we perform the initial configuration and manage them going forward? First, it's time to check in with Sally, Sue, and John.

Globomantics Architecture Updates

Our deployment is shaping up nicely. We've got a multi-zone, multi-instance application running up in AWS. But, of course, nothing in IT is ever really done. The Dev and Ops teams both have new requests. Our friend Sally Sue has a couple requests from the development side of the house. For starters, she would like to give us the website files and have them dynamically uploaded to the web servers at startup. She would also like to get access to the request logging from the load balancer for analysis and debugging. John has a few things he'd like to see us implement as well. As Terraform is adopted by Globomantics, he wants to make sure we are all using the same major version of Terraform and the provider plugins. He would also like the Terraform files to be formatted consistently to help with sharing across the teams. Why don't we start with Sally's two requests by updating our architecture to support her needs. It sounds like she needs an S3 bucket for logging, and we can also put her website files there to be picked up by the EC2 instances when they start up. In our architecture, we will add an S3 bucket and upload the website content to it. Then we will assign the EC2 instances a profile that has access to copy information from the S3 bucket. The load balancer configuration supports logging to an S3 bucket, so we can use the same S3 bucket to write those access logs out. S3 buckets need to have globally unique names, and that's something we can generate with the random provider for Terraform. With that in mind, let's see how we can add a provider and meet some of John's requests.

Terraform Providers

Provider plugins are Terraform's superpower. We talked about providers in an earlier module, but now that you've had a chance to use them, it's time to go into greater detail. As we have already seen, Terraform provider plugins are available in the public registry at registry.terraform.io, but you can also get provider plugins from other public registries, privately hosted registries or even your local file system. We aren't going to get into that use case in this course, but it's useful to know. There are three types of provider plugins available on the Terraform hosted registry, Official, Verified, and Community. Official providers are written and maintained by HashiCorp. Verified plugins are written and maintained by a third-party organization that has been verified by HashiCorp and is part of the HashiCorp technology partner program. Community provider plugins are written and maintained by individuals in the community, and have not gone through the verification process. There's nothing inherently wrong or bad about using community providers, but you should be aware of their providence and probable level of support. One thing all the providers have in common is that they are open source and written in Go. If you have the inclination to inspect or contribute to a provider, the code is readily available for you to do so. Providers themselves are a collection of data sources and resources, as we have seen when reviewing the documentation. Providers are versioned using semantic version numbering. You can control what version of a provider plugin you use in your configurations so you can avoid a situation where a provider is updated and it breaks something in your deployment. Within your configuration, you can invoke multiple instances of the same provider and refer to each instance by an alias. For example, an instance of the AWS provider is limited to one region and account. If you wanted to use more than one region in a configuration, you could do so with multiple instances of the AWS provider and aliases. Let's take a look at how you can

specify the source of a provider plugin and the desired version. This will help us fulfill John's request.

Provider Syntax

We have already seen how to create a provider block in our configuration, and you might assume that is where you would specify more information about the provider, like its version and source. That assumption would be well founded, but unfortunately incorrect. Provider information is defined in the terraform configuration block using a nested block called required_providers. We haven't seen the terraform block before. It is used to configure general settings about a Terraform configuration, including the version of Terraform required, back-end settings for the state data, required provider plugins, provider metadata, and experimental language features. For the purpose of this course, we will focus on using the terraform block to define our required_providers. Each key in the required_providers block will be the name reference for a provider plugin. The convention is to use the standard provider_name, unless you're going to have multiple instances of a plugin from different sources. That's an advanced topic and one you're unlikely to encounter when you're first getting started with Terraform, so don't worry about it right now. The value for the provider key will be a map defining the source of the plugin and the version of the plugin to use. By default, Terraform assumes you are getting your plugin from the HashiCorp-hosted Terraform Registry, and it provides a simple shorthand for the address value. There is an expanded form of the address for alternate locations. The version of the provider can use several different arguments, including setting it equal to a version, a range of versions, or using a special sequence of a tilde followed by a greater-than symbol. That last one is not immediately intuitive, so let's look at an example. We've been using the AWS provider from the Terraform Registry. If we wanted to add it to our required_providers block, we would set the key to aws, as that is the name of the provider in the documentation. For the source, we can use the shorthand of hashicorp/aws. Since we're not giving Terraform a full address of the plugin, it will try and find the plugin on the Terraform Registry. Under our version, let's say we wanted to stay on version 3 of the plugin, but we don't care about the minor version. Using the tilde and greater-than symbol tells Terraform to find the latest plugin that is of the form 3.x. If we wanted to stay on the minor version of 3.7, we could update the expression to 3.7.0, and that would keep us on the latest 3.7 release. Most of the breaking changes in a provider will come from a major version release, so staying on the same major release of a plugin should keep things stable, although your mileage may vary depending on the provider. Once we've defined our required_providers, we can reference them in a provider block. The block starts with the provider keyword, and then the name of the provider used in the required provider block. If you are going to create more than one instance of the provider, you can add an alias argument inside the block, providing a string for that instance of the provider. And then you can provide any additional arguments that are specific to that provider. Assuming we've gone with the convention and used aws as the provider_name for the AWS provider, our provider block stays the same, with aws as the name label. If we want to create an additional instance of the AWS provider for a different region, we could give an alias of west in the block. To use the aliased instance of the provider with a resource or data source, we would add the provider argument to the configuration block. The value would be the provider_name.thealias, which would be aws.west, in this case. If no provider argument is specified, Terraform will use a

default provider instance with no alias set. Armed with all this new knowledge, let's head over to our configuration and add a required_providers block for the AWS and random providers.

Specifying Required Providers

Before we add the terraform and required_providers block to our configuration, let's take a look at the docs. We can browse to the AWS provider by clicking on Browse Providers and going to AWS. And let's go into the Documentation tab, and in the beginning of the AWS Provider it provides some Example Usage, and there is the terraform block and required_providers block. That's exactly what we want, so let's go ahead and copy that text from the example. So I'll copy that text, and let's go over to our configuration. In our configuration, let's go ahead and expand the globo_web_app directory, and we're going to create a new file called providers.tf, and in the providers.tf file we'll go ahead and paste that text. Okay, now we are sourcing our provider from the public registry, and we're setting the version to stay on major version 3. Right now 3.63 is the latest version, but when they come out with 3.7, we'll automatically upgrade to 3.7. If version 4 comes out, we will not automatically upgrade to that, and that is what we want. Now that we have our AWS provider added, there's something else I want to point out about the AWS provider documentation. Something that we glossed over when we were looking at the documentation earlier is the Authentication section for the AWS provider. This provides information about how to authenticate using the provider. We've been using static credentials up until now, defined in variables. There are many other options for authentication. We have environment variables; a shared credentials or configuration file which is generated by the AWS CLI; and also if you're running Terraform on AWS, you can leverage CodeBuild, ECS, EKS or the EC2 Instance Metadata Service. Instead of using hard-coded credentials with variables, an approach that I've often seen is using environment variables. Let's scroll down to the Environment Variables area. We can provide our credentials with two environment variables as opposed to defining variables inside of the Terraform configuration. By doing that, we will prevent someone from accidentally hard-coding credentials in a terraform.tf vars file and checking that into source control. That's bad. We don't want that to happen. So let's go back to our configuration, remove those variables, and from here on out we can use the environment variables instead. Okay, back in the configuration, let's go ahead and open up our variables file, and we are going to delete the aws_access_key and aws_secret_key from our list of variables. We'll go ahead and do that now, there we go. We're going to keep the aws_region, because we need to define that for our provider. I'll go ahead and save that file, and now let's go to where our provider is defined. Right now that's sitting in the network.tf file. That's probably not the best place for it, so let's go ahead and remove that from the network.tf file, and instead we'll add it to the providers.tf file. So I'll go ahead and add it in there, and now I can remove the access_key and secret_key arguments, since we'll be supplying those values through the environment variables defined in the documentation. Now let's head back to the documentation and we will walk through adding the random provider to our configuration.

Adding the Random Provider

We are going to use the random provider and the random_integer resource in the provider to help generate a unique ID for our S3 bucket. Remember S3 bucket names need to be globally

unique. The easiest way to do that is add some sort of unique ID to the end of the name for your S3 bucket. Let's go ahead and search for the random provider, and it comes up as the first result. We can go ahead and click on that, and if we want to see how to use the provider, we can actually click on the USE PROVIDER drop-down. That will give us an example of how to add it to our existing terraform block or add a new terraform block if we don't have it. My challenge to you now is to add this additional provider to the required_providers block in the configuration and set the version to stay on major version 3, but accept updates in the minor version. Go ahead and pause the video now and try to add the provider on your own. Okay, let's see how you did. Going back to the configuration, I have added the random provider to my required_providers block, and I've set the version to ~> 3.0, which will keep us on the major 3.0 version. The resource we want to add from random is the random integer, so let's go back to the documentation and see how we add that. Back on the website, let's click on the Documentation area, and before we expand the resources, one thing I want to point out about the random provider is that it doesn't have any configuration options for the provider block, which means you don't actually need to include a provider block in your configuration since there's nothing to configure. With that in mind, let's expand the resources here and take a look at the random_integer. Here's the random_integer with an example usage. What I would like for you to do now is add the random_integer resource to the locals.tf file in our configuration, and set the minimum to 10000 and the maximum to 99999. You don't have to include the keepers argument, just a min and a max and use the name_label of rand. Go ahead and pause the video now, and we'll come back and see how you did. Okay, let's go to my updated configuration and see how I added the random_integer resource to my locals.tf file. Here is my locals.tf file, and you can see I've added the resource random_integer with the name_label rand, setting a minimum value of 10000 and a maximum value of 99999. When we use this as part of our S3 bucket naming, we will have a five-digit random integer that will be appended to the name of the S3 bucket. With those components in place, let's look at the resources we need to add for our S3 bucket and to allow access from the EC2 instances.

Creating IAM and S3 Resources

With our provider situation figured out, we can turn to Sally's request to add an S3 bucket for website content and logging. Before we jump back into the configuration, let's figure out what resources we will need to create. We are going to create an S3 bucket and place objects in that bucket for the website. To accomplish that goal, we are going to use the aws_s3_bucket resource and the aws_s3_bucket_object resource. Our EC2 instances will need access to the S3 bucket, but we don't want to make the bucket public for everyone, instead we can create some IAM resources to help us grant access to the EC2 instances. We'll create a role using the aws_iam_role and grant that role permissions to the bucket with an aws_iam_role_policy. Then we can assign the role to the EC2 instances by creating an aws_iam_instance_profile and then adding an entry to our aws_instance block to use that instance profile. We also need to provide the load balancer access to the S3 bucket, and we can do that through a bucket policy that refers to a data source of aws_elb_service_account. That will give us the service principal account for the elastic load balancer in the region that we're currently working in, and we can grant that access to the S3 bucket. With all that context in mind, let's head over to our configuration and add some placeholders for each resource. Back in our configuration, let's add a file for the

S3 configuration and we'll call it s3.tf, and within that file let's add placeholders for all the different resources that we need to create. So I'll add in the comments for the file, aws_s3_bucket, aws_s3_bucket_object, aws_iam_role, aws_iam_role_policy, and aws_iam_instance_profile. In addition to these resources, let's go ahead and open up the loadbalancer file, and let's add a placeholder to the beginning of this loadbalancer file. Okay, there we go, we've got all of our placeholders. For the website files that we'll be uploading to our S3 bucket, those are located in the root of the exercise files. So we can scroll down to the bottom here. Those are the website files. We've got an index.html file and an image file of the Globomantics logo. We'll go ahead and copy this website directory and paste it in our globo_web_app directory. There we go, we now have our website files. The other thing we need to do is create a name for our S3 bucket, and we can do that by defining a new local value. So let's open our locals.tf file, and we'll add another value in here. We'll set the name to s3_bucket_name, and we'll set the value to globo-web-app-, and then the integer from our random_integer resource, which we can do by adding the interpolation symbol with the dollar sign and the curly braces, and then the reference to the result from that resource, which is going to be random_integer.rand.result. That will append a five-digit unique ID to our S3 bucket name. Going back to the s3.tf file, my challenge to you, if you really do want a challenge, is to try to add all of the necessary resources to this file. I will say that many of these resources require you to write an IAM policy or a bucket policy, and that's very difficult, so maybe skip those portions of the resource configuration and try to do the rest, along with the loadbalancer data source. And then you can take a look in the M6 solution directory to see how the IAM policies are configured for each of the resources that uses it. This is going to be a real challenge, so if you want to do it, go ahead and pause the video now and we'll come back to see my solution.

Viewing the Updated Configuration

Okay, welcome back. Let's see how you did. And first, we'll start with the S3 bucket. The bucket argument is going to be the name of the bucket, and we're going to set that to our local value. The acl Is going to be private because we don't want this to be a public bucket. We'll set the force_destroy = true, which allows Terraform to destroy the bucket. Now below that is the policy, and we're going to embed the entire policy here, which is in JSON. In order to do that, we are going to use the heredoc syntax that we saw when we configured the user data for our instances. And this will replicate this text exactly except for the interpolation that we've added for the values from Terraform. So let's take a look at what's in this policy. And don't worry about being a bucket policy or an IAM policy expert. This is a Terraform course, after all, and not one on AWS. So I'm just going to point out the relevant things for you if you're ever writing one of these policies. In our statement for the bucket policy, we want to allow the load balancer and the delivery logs service access to this S3 bucket. We do that by adding an effect of allow, and we're going to reference a principle here from our Elastic Load Balancer service account data source. So if we go over to the loadbalancer file, this is the data source that you'll need to reference the service account used by Elastic Load Balancers in your region. Okay, going back to the S3 file, for the action, we're giving it s3:PutObject, and for the resource, we're giving it the bucket name and then the path, alb-logs. This gives the Elastic Load Balancer permission to write data to that path in our S3 bucket. We're also going to give that same permission to the service delivery.logs.amazonaws.com. And we're going to give that service an additional permission of s3:GetBucketAcl. This entire policy is available

on the AWS docs, so don't worry about trying to memorize it or anything. You can always go back to the documentation and find it. Alright, scrolling down to the next resource, we have our two bucket objects, which are the website components we want to upload to the S3 bucket. We first have the bucket argument that references the S3 bucket, and then we have a key, which is the destination on the S3 bucket where it should create that object, and the source is where to get that object from. We're getting it from the website directory that we copied into our configuration directory earlier. Scrolling down a bit more, we get into the IAM portion of things by first creating the IAM role that's going to be used by our instances. The name is allow_nginx_s3. And for the assume role policy, this allows EC2 instances to assume this role. That's all that does. Scrolling down a little bit more, we get into the role policy, and this is the policy that actually grants permissions to access the S3 bucket. We're naming it allow_s3_all, and we're assigning it to the role that we just created by name, and then we define the policy with the same heredoc syntax. We're giving it the Action s3:*, which means you can do anything in the S3 bucket, and we're assigning it the resource of the bucket name and any paths along that bucket name. That's the policy that's assigned to the IAM role we just created. Scrolling down a bit more, we get into the instance profile. The instance profile is what's going to be assigned to the EC2 instance. We're giving it the name nginx_profile. We're associating it with the role that we created earlier, and we're giving it the common tags like we have with everything else in this configuration that supports common tags. The next thing we need to do is update our instance and our load balancer to take advantage of this S3 bucket. But before we do that, we need to talk about dependencies.

Planning and Dependencies

When Terraform is trying to make the deployment match your configuration, it has to run through a planning process. Terraform goes through this process when you run a plan, apply or destroy. As part of the planning process, it needs to figure out the order in which to create, update or delete objects. To calculate a plan of action, Terraform will first refresh and inspect the state data. Then it will parse the configuration and build a dependency graph based on the data sources and resources defined in the code. Comparing the graph to the state data, Terraform will make a list of additions, updates, and deletions. Ideally, Terraform would like to make the updates in parallel, so it tries to figure out which changes are dependent on other changes. Changes that are not dependent on other changes can be made at the same time, while changes that have a dependency will have to be done serially. How does Terraform figure out the order in which changes need to happen? References. Let's look at an example that we have in our configuration right now. In our current configuration, we are creating a VPC, a subnet, and an EC2 instance. The VPC doesn't refer to any other resources in the configuration, so Terraform can create it immediately. If we look at the arguments in the aws_subnet resource, we have a reference to the vpc.id. The reference creates a dependency on the VPC resource. Terraform will wait until the VPC is created, and then use the ID to create the subnet. Our aws_instance configuration has a reference to the aws_subnet ID. That creates a dependency for the EC2 instance, so Terraform will wait for the VPC and then the subnet to be created before it tries to create the EC2 instance. Terraform can infer the dependency tree for this configuration implicitly. It doesn't need you to tell it that the subnet is dependent on the VPC. Sometimes a dependency is non-obvious, and you must explicitly tell Terraform about it. We actually have that situation in our configuration right now, you just don't know it yet. It's one of those things that you only figure out once it breaks, so let me

explain. In our configuration, we are creating an aws_iam_role. Both the instance_profile and the role_policy directly reference the iam_role, so Terraform will wait until the role exists to create those two resources. To assign proper permissions to our EC2 instances, we have to add the instance_profile to our aws_instance configuration, which creates a dependency between the instance_profile and the instances. However, in order for the EC2 instances to actually access the S3 bucket using the iam_role_policy, it also needs to be created. If the EC2 instance starts up before the iam_role_policy is ready, access to the bucket will be denied, and that's bad. The solution is to add a depends_on argument to the aws_instance resource that references the iam_role_policy. With that explicit dependency, Terraform will wait until the iam_role_policy creation is complete before moving on to creating the aws_instances. Generally speaking, Terraform is pretty good at detecting implicit dependencies. The depends_on argument should be used sparingly, and only when an explicit dependency is required. Armed with that knowledge, let's go back to the configuration and update the load balancer and EC2 instances.

Updating the Load Balancer and Instances

Okay, my challenge to you is to go into the load balancer and add the access log configuration and go into the EC2 instance, add the instance profile, and add that depends_on argument. The depends_on argument is expecting a list of references to other resources within the configuration. So go ahead and try that now, pause the video, and when we come back, we can take a look at my updated configuration. Alright, welcome back. Let's see how you did. Let's first take a look at the load balancer configuration. In the load balancer configuration, you can see there's now an access_logs configuration block inside of the resource, and in there, we are referencing the bucket that we created. We're going to use the actual reference and not the bucket name, so we create a dependency between the load balancer and the bucket. The prefix is going to be alb-logs, and enabled is set to true. We want to write logs there. Okay, now let's take a look at the instances. In our instances, I have added an argument, iam_instance_profile, and it's set to the name attribute of the profile that we created. For depends_on, we're giving it a list of resources it should be dependent on, so we need the square brackets, and then in there, I am referencing the iam_role_policy.allow_s3_all. So the instance will wait until that role policy is created before spinning up the instance. If we scroll down a little bit, we also have to update our second AWS instance, so don't forget to do that. We want them to be configured the same. The last thing we need to do is update our user data script. But before we do that, let's discuss post deployment configuration options.

Post Deployment Configuration

After a resource is created, sometimes you need to perform post-deployment configuration. It could be loading an application onto a virtual machine, configuring a database cluster or generating files on an NFS share based on resources that are created. If you want to stay in the Terraform ecosystem, there are many providers and resources that can help you with post-deployment activities. If you want to create a file, there's a file resource. If you need to configure a MySQL database cluster, there is a MySQL provider. Using native Terraform resources will often be the answer. Another option specific to servers is to pass data

as a startup script to the server operating system. All the major cloud providers offer a way to pass a script, although the name of the argument changes. For AWS, we are already using the user data argument to pass a startup script. The downside to passing a script is that Terraform has no way to track if the script is successful or not. It's simply another argument in the configuration. If the script fails, you need to gracefully handle that, or you could go outside of Terraform and leverage configuration management software. There are many different config management options out there, which Terraform can hand off to for post deployment configuration. Ansible, Chef, Puppet are three well-known examples. A common practice is to bake the configuration management software into a base image for a machine and have Terraform use that base image when it creates an instance. If all else fails, you can use Terraform provisioners. You're likely to encounter these out in the wild as you ramp up on Terraform, so let's dig into what provisioners are and why they're usually a bad idea. Provisioners are defined as part of a resource, and they are executed during resource creation or destruction. A single resource can have multiple provisioners defined with each provisioner being executed in the order they appear in the configuration. If you need to run a provisioner without a resource, there is a special resource called the null_resource that allows you to run provisioners without creating anything. If a provisioner fails, you can tell Terraform to either fail the entire resource action or continue on merrily. Which one you choose will depend on what the provisioner is doing. HashiCorp considers provisioners as a last resort when all other options have been considered and found lacking. Provisioners are not creating objects Terraform can fully understand and manage, which puts the onus on you and your team to ensure things like error checking, idempotence, and consistency are implemented properly. There are three provisioner types. The file provisioner will create files and directories on a remote system. The local-exec provisioner allows you to run a script on the local machine that is executing the Terraform run. Local-exec is used as a workaround for functionality that may not yet be in a provider, and it's probably the provisioner you'll see most often. Remote-exec allows you to run a script on a remote system. Most of the time, the file provisioner and the remote-exec can be easily replaced with a startup script through something like user data. There used to be more types that were specific to configuration management products like Chef or Puppet, but all of those have been deprecated. In case you encounter provisioners out there in the wild, let's look at some examples of how they're configured.

Provisioner Syntax

In the file provisioner example, we are first defining how the provisioner can connect to the remote machine to copy those files. It is also possible to define a connection block for all provisioners used in a resource. The connection types are either going to be SSH or WinRM. A provisioner can refer to the attributes of the resource it lives in using the self object. For instance, here we are getting the public IP attribute of an EC2 instance the provisioner needs to connect to. The source and destination arguments define the files or directories that should be copied to the remote machine. The local-exec provisioner does not need a connection block since it is running on the local machine. You can pass it a command to execute and specify which interpreter to use for executing the command, for instance, Bash, PowerShell, Perl or any other interpreter that you have. The remote-exec provisioner will need connection information defined in the resource or in the provisioner. Remote-exec can execute an inline script, a script stored in a file or a list of paths to local scripts executed in the order they are

provided, which is what I'm showing here in the example. As I said, HashiCorp recommends heavily against using provisioners whenever possible, but you still may encounter them in your Terraform travels. For our configuration,we're going to stick with user data for post-deployment config. Let's head over to our configuration and update it to grab those website files from the S3 bucket onto those EC2 instances.

Updating the Startup Script

Our goal here is to update the script that's defined in the user data argument for each instance. Were trying to grab the two files that make up our website from the S3 bucket, copy them down locally, and then move them to the /usr/share/nginx/html directory. So we're going to replace some of the commands that are here with new commands. The good news is Amazon Linux comes with the AWS CLI, so we can use the AWS S3 commands that are baked into the CLI, and the command line will automatically use the instance profile that's been associated with the EC2 instance to authenticate to the S3 bucket. Now if you'd like to, you can pause the video now and update the command to copy those files over. And when we come back, I will show you the updated script that I have. Okay, here's my updated script. We're using the aws s3 cp command to copy two files from the bucket to our home directory, that's /home/ec2-user, and then we are removing the default index.html file from the nginx installation and copying the files from our home directory over to that nginx HTML directory. With our configuration complete, let's step into the next phase, which is to get this configuration validated and deployed.

Formatting and Deploying the Updated Configuration

One of the things that John from the ops team asked us to do is to make sure that our files are formatted properly, and we can do that by using the terraform fmt command. So I'll open up the terminal window now, and terraform fmt works on any files it finds in the current directory that you run it from. So if we run terraform fmt, it will look at the current formatting for each of the files in the directory and then make updates to those files to bring them in line with HashiCorp standards for HashiCorp configuration language files. If you're curious about what has changed in those files, go ahead and open up and inspect those files, and you can see how the formatting has changed. The next step in our process is running terraform init again. You might be wondering why. And the reason is because we added a provider to our configuration, and Terraform needs to download that provider plugin from the Terraform registry. So let's go ahead and run terraform init now. Okay, if we scroll up a little bit, we can see it installs the HashiCorp random version 3.1.0. It's going to continue to use the previously installed AWS plugin because our updated version setting doesn't change which version is installed. Okay, so now that we have initialized our configuration, the next thing to do is validate our configuration. So we'll go ahead and run terraform validate. And excellent! My Terraform configuration is valid. You may get some errors, so go ahead and pause and remediate those errors now, and then we'll resume by running terraform plan. Okay, remember that we removed our AWS access and secret key from the variables, so we now need to set them as environment variables. If we expand the commands directory and open up m6_commands, here are the commands for Linux and macOS or for PowerShell to set the proper environment variable for the AWS access key and secret access key. Go

ahead and update those values and run the command to set your environment variables. I've already run those to set my environment variables. Once we have those environment variables set, now we can run terraform plan, and we'll send the output plan to m6.tfplan. I'll go ahead and run that now. And based on the plan, we have 11 things to add, 1 to change, and 4 to destroy. If we maximize the view here, we can get an idea of what it's creating. We know we're creating a bunch of resources because we added them, but I'm curious to see what is changing or being destroyed. So let's scroll up, and we see that the target_group_attachment is being replaced because the target_id is being replaced, which tells me that the instances are also being replaced. If we scroll up to one of the instances, we can see the instance is being replaced. And if we scroll down with the instance, the user data has been updated, which forces a replacement of the EC2 instance. It's interesting to note that adding an IAM instance profile does not require a replacement of the EC2 instance, so it's actually updating that user data that is forcing the replacement. Lastly, the load balancer is being updated in place because we have updated the access logs configuration. That doesn't force a replacement. We're just updating as is. Alright, all that sounds good. Let's go ahead and run terraform apply "m6.tfplan", and that will apply the changes that were listed in the plan. This will take a few minutes to recreate those AWS instances, so I'll go ahead and pause the recording now and resume when the deployment is complete. Okay, my deployment is complete, although I had to run it a second time because the nginx profile I was trying to create already existed. So I had to delete it and then let Terraform recreate it. So pro tip, make sure you don't already have a profile named nginx_profile. Now that the deployment is complete, let's go ahead and go to the address so we can generate some traffic on our website, which will then cause the load balancer to write data to the access logs. So I'll go ahead and grab this address. And in our browser, I'll go ahead and open up a new tab. And now you can see our updated web content is being loaded by the EC2 instances. We've removed the ability to differentiate between the two different servers since it's not really necessary anymore. We know that that works. I'll go ahead and refresh the website a few times just to generate some web traffic that will be written to the S3 bucket. Okay, now that we have generated some traffic for our website, we can go over to the S3 console. Here's the S3 bucket that we created using Terraform. In there, we can see we have two paths. We have alb-logs and the website. Let's go into the alb-logs. And there we have a folder, AWSLogs, and there we have one based off of our account. And in there, there is a test file that was run when we updated our configuration of the Elastic Load Balancer. It may take 5 or 10 minutes for the load balancer to process new requests and add them to the access log for the S3 bucket. So if you don't see access logs right away, don't worry. They will be there shortly. At this point, we have met all the requirements from both the development team and the ops team. Good job!

Summary

In this module, we learned how to add a new provider to a configuration, and we also saw how to properly specify the version and the source for our provider using the required provider's block. We updated the architecture for our configuration to include an S3 bucket, and in the process, learned about Terraform's dependency graph. Lastly, we talked about why provisioners are a bad idea and other options for performing post deployment configuration. The next step in evolving our configuration is to add functions and looping into the mix. Looping helps us create multiple instances of an object efficiently and dynamically, and

functions can help us transform data in our configuration to make it more useful and effective. That's coming up in the next module.

Using Functions and Looping in Your Configuration

Overview

Terraform has some more tricks up its sleeve when it comes to creating a dynamic and efficient configuration. A key feature of any programming language is the ability to create loops and use functions, and Terraform is no exception. Hey everyone, this is Ned Bellavance. I'm a HashiCorp ambassador and founder of Ned in the Cloud. Let's get loopy adding functions and iteration to our configuration. We'll kick off this module with some new ideas from our old buddy, John. He's been reading up on iteration and functions in Terraform, and he has a few ideas to improve our configuration. That means it's time to do some learning of our own. We'll check out what looping constructs exist in Terraform and how they can be used to make our config more dynamic and flexible. We're also going to want to use some functions to fulfill John's requests. We'll see what type of functions are available, how they're used in a configuration, and how to test expressions using Terraform console. First, let's check in with John and see what suggestions he has.

Globomantics Updates

We fulfilled the requests from Sally Sue when it comes to the deployment architecture, but now John has a few suggestions on how our code could be more effective and efficient. To start with, John would like to be able to dynamically increase the number of instances deployed for the application. Two instances might be good for development, but in a production scenario he'll likely need more. He would also like to decouple the startup script from the configuration files and store it in its own file for possible updates and reuse. John also thinks it's a little cumbersome to set CIDR addresses for the subnets and the VPC. He'd like to be able to just set the VPC CIDR address and let Terraform split it up among the subnets. Finally, he's noticed that we've been a little inconsistent with our naming of AWS resources. He'd like to be able to add a naming prefix and apply it consistently across all resources. You tell him, not a problem. Terraform and you can take care of it. The updates that John requested aren't going to change our architecture. The goal is to keep the deployment the same while improving our infrastructure as code. Let's start by checking out the looping constructs in Terraform.

Loops in Terraform

Terraform has several different ways to create multiple instances of an object or manipulate collection objects. We'll start with an overview of the various options and then drill down into the two that are most useful for our configuration. The first looping construct to consider is the count meta-argument for modules and resources. Count is used to create multiple instances of a resource or module when the instances are very similar in nature, the value for

a count argument is an integer, and that includes 0. You can tell Terraform to create 0 of a resource by setting the count to 0, which sounds like an odd thing to do. It's actually super useful when you want to make the creation of a resource conditional on other factors in the configuration. The next construct is the for_each meta-argument, which is also used for modules and resources. For_each takes a set or a map as a value. It's used instead of count in situations where each instance will be significantly different than the others. You have full access to the values stored in the set or map you submit, and those values can be used when configuring each instance of the resource. That gives you a lot more flexibility than a simple count integer. Dynamic blocks are used to create multiple instances of a nested block inside a parent object. They accept a map or a set for a value to construct the blocks. This is an advanced topic that we're not going to cover in this course, but I included it for completeness. Let's focus in on the syntax of the count and for_each meta-arguments, since we will be using both in the configuration.

Count Syntax

The count meta-argument can be used for resources or modules. The syntax for either is the same, and since we haven't touched on modules yet, we're going to use resources for our example. The count argument goes inside the resource and accepts an integer as a value. The integer determines how many instances of the resource should be created. In our example, the count is set to 3, so Terraform will create three EC2 instances. When the count argument is used, a special new variable is available called count.index. As Terraform loops through the creation of each instance, count.index will resolve to the current iteration Terraform is on. You can use this value anywhere in the resource configuration block. In our example, we are using count.index to name our EC2 instances globo-web-, the number of the iteration. Count starts at 0, making the first instance globo-web-0. Now you might be wondering, how do I reference the instances that I'm creating with the count argument? Well, the count argument is going to create a list of resources. Each element of the resource list can be referenced by a number. The syntax is similar to standard resource addresses. We start with the resource-type.name_label, and then we add a square bracket with the element number of the instance we want, optionally followed by the attribute of the instance if needed. In our example, if we wanted to refer to the name attribute of the first AWS instance, the syntax would be aws_instance.web_server[0].name. If you would like to get an attribute of all of the instances, you can use an asterisk in the square brackets. This will return a list containing the attribute value for each instance.

For_each Syntax

The for_each meta-argument can also be used in resources or modules. The value for the for_each argument will be either a set or a map. As a quick reminder, a set is an unordered collection of objects. A tuple and a list are ordered collections, so you cannot use a list or a tuple directly, but you can transform a list or a tuple with the toset function. In our example, we are using a map with a set of key-value pairs. Terraform will look at the number of elements in the map or the set and create a corresponding number of instances. In this example, we have two entries in the map, so Terraform will create two S3 bucket objects. In a for_each loop, there are two special variables, each.key and each.value. During the looping

process, each.key will be set to the key of the map item currently being iterated over. What about each.value? I think you can probably guess what it's set to, the value corresponding to the current key. If you are iterating over a set instead of a map, each.key and each.value will be equal to the same thing. Values in the map or set do not have to be a primitive data type, like a string or a number. It could be a complex object with nested values that you'd like to use in each iteration of the resource. Using a for_each argument is going to create a map of resources. Each entry in the map can be referenced by the key name, just like we've seen when dealing with map data types in the past. The syntax is the resource type, followed by the name_label, and then square brackets with the key string in quotes, followed by dot and the attribute you're interested in. In our example, if you wanted to get the id attribute of the cheese instance, the syntax would be aws_s3_bucket_object.taco_toppings["cheese"].id. Just like the count syntax, if we want to get the id attribute of all of the instances, we can swap out the key string with an asterisk. The returned value would be a list of all of the IDs. Based on these two looping constructs, let's see if we can find some places in our configuration that would benefit from using count or for_each.

Looping Targets

Within our configuration, we should be on the lookout for anywhere we are creating more than one of the same resource. If you'd like to look through the config now and make some guesses, feel free to do so. Here's the list that I came up with, starting with the primary resources that we can update. We have two AWS subnets right now and possibly more in the future, depending on how the architecture evolves. Each subnet is almost identical to the others, except for the CIDR address and availability zone, which makes them a good candidate for the count loop. Likewise, we are creating multiple EC2 instances that are fairly undifferentiated, except for the subnet they attach to. Looks like we'll be using a count loop for them as well. Lastly, we are creating multiple AWS S3 bucket objects, but those have different names and paths, so it might make more sense to use a for each loop to create them. Since we are going to use loops to create these resources, there are going to be other resources that will be impacted as well. We need to create an AWS route table association for each subnet, so we can use a count argument there. We also need to create an AWS load balancer target group attachment for each EC2 instance, so we'll use a count argument there as well. Let's jump over to the configuration and set a few things up.

Updating the VPC and Instances

Let's start by opening our network.tf file. I'll go ahead and expand the directory and open network.tf. All right, there we go. And let's scroll down to the definition for our subnet. Scrolling down to the first subnet, we are going to update this first subnet resource for all of our subnets by adding a count argument. Now, what's going to drive that count argument? Let's first set up a variable to define how many subnets we're going to create with this resource. Let's open the variables.tf file. All right, there we go. And we are going to add a value. Let's go ahead and add it below the vpc_cidr_block. We'll call the variable vpc_subnet_count, we'll set the type = number, we'll set the description to the number of subnets to create, and we'll set the default = 2. Now that we have our variable ready, let's go back to the network.tf file and update the resource block. For the resource block, let's change

it from subnet1 to subnets. This is going to create all of our subnets after all. And below there, we will add our argument for count, and we'll set the value of the count = vpc_subnet_count. Next we need to update our CIDR block, and we can use the count.index to select an item from the vpc_subnets_cidr_block variable, so we'll set this to count.index. On the first iteration, it will select the first element from the list, and on the second iteration, it will select the second element from the list, and so on. The vpc_id will remain the same. They're all in the same VPC. The map_public_ip_on_launch will remain the same. The availability_zone will also need to update with the count.index to select the element from the names list. There we go, we've updated the value to count.index. That's everything we need to change for this resource. Below it, we have our subnet2. We no longer need subnet2 because we're defining all of our subnets with that single resource block, so we'll go ahead and delete this resource block. There we go. Now the other thing we need to update our the route table associations for the subnets. Let's scroll down to that resource. Let's rename our first route table association resource rta-subnets. And now we'll add the count argument to this resource. We'll set the count equal to the number of subnets because that's how many route table associations we need to create. Now we need to reference each subnet that we created with our subnet resource. We'll use the resource addressing that we learned earlier to create that reference. So it should be aws_subnet, and remember, we changed the resource to subnets. And we want to specify a particular subnet, so we'll add the square bracket, and within the square bracket we'll add count.index. This way, in the first iteration, it will reference the first element in the list of subnets and the id attribute. And then on the second iteration it'll reference the second subnet, and so on. The route table id stays the same because we're associating all these subnets with the same route table. Now that we've updated this resource, we can delete the rta-subnet2 resource. And that takes care of updating our subnets and the route table association. My challenge to you is to update the instances with a count argument as well. In the instances file, you can update the first instance to nginx instances or whatever name label you would like to use, and add a count argument. You'll need to add a variable like instance_count for the number of instances that will be created, and within the configuration, you're going to need to reference the proper subnet for each instance. That's going to end up being a little more complicated than you would initially think so. For now, we can safely assume that we just have the two instances and two subnets, one instance per subnet. The other thing you'll have to update is the target load balancer group attachment, and that is in the load balancer.tf. Down at the bottom we have our two target group attachments. You're going to update it so that there is only one that is also using the count argument and referencing the proper target IDs to the instances that you're creating with the loop. So go ahead and try to do that now, and when we come back you can see my updated configuration. Okay, welcome back. Let's see how you did. First, I added a variable for the instance_count, and I added it right below instance_type to kind of keep the same variables together. Instance_count is set to type number, and the default is equal to 2. Now let's check out the instances.tf file. For instances.tf I renamed the resource to nginx instead of nginx1. The count is set to var.instance_count, and the subnet_id reference I updated to subnets and then the count.index. We're actually going to change that a little bit in the future, but for now it's okay to leave it like that. Under loadbalancer I set the count to var.instance_count, and for the target_id I updated the reference to nginx and the count.index for that element out of the list of instances.

Updating the S3 Bucket Objects

Now the last thing to update is our bucket objects. So let's go into s3.tf, and we are going to update the bucket objects to be a single bucket object. We'll start by updating the name label to website_content. Then, we will add a for_each meta argument, and we're going to use a map for our for each. So I'll set the curly braces to indicate a map. The first item in the map, the key will be website, and we'll set it to the path of the website file that we want to upload to our S3 bucket, which would be /website/index.html. The second item will be the logo, so I'll set the key to logo. And then the path to the logo is /website/Globo_logo_vert.png. We're going to use the same bucket as the target for each bucket object, so we can leave that the same. The key, which is the path for the object on the S3 bucket, we can set that to each.value, which will use the value that's stored in each map key. And then the source is the path to the file that we want to create as an object. We're going to use the current directory by specifying dot, and then we'll use the interpolation syntax to set it to each.value. So on the first loop, this will evaluate to ./website/index.html. And that way we will create all of our AWS bucket objects. We can also delete the second resource here because we no longer need it. You could make this more dynamic by creating a variable that includes all of the items that need to be uploaded or even use a function of some kind to evaluate all of the files in a directory. If you'd like to do that, I leave that as an exercise to you. For now, we're going to leave this as hard coded values in the for each statement. Now speaking of functions, we are going to need to use functions to meet the rest of the requirements that John has laid out for us, so let's dig a little deeper into functions and expressions within Terraform.

Terraform Expressions and Functions

Terraform includes functions and expressions to support the manipulation of data in HCL files. We've already seen the expressions and even some of the functions at work, but now it's time to examine them in more detail. We've been using Terraform expressions for a while now, in particular, the interpolation and heredoc expressions to include resource and variable values in a string or pass an entire string to an argument like user data. Terraform also supports arithmetic and logical operators like and, or, equals, greater than, etc. The evaluation will depend on the data type you are operating on and whether that data type supports the comparison. Terraform also supports conditional expressions, which are essentially an if statement followed by a value to return if true and a value to return if false. You can combine a conditional expression with a count argument to decide if a resource is created or not. The for expression is used to manipulate and transform collections. It can take any collection object type, map, list, set, etc., and it will return a new list or map. For expressions are a great way to work with the set of instances that a count or a for each argument generates from a resource block. Just like any other programming language, Terraform supports functions that help you transform and manipulate data. Unlike provider plugins, functions are built into the Terraform binary, so you don't have to initialize or download anything to use them. Since they don't use an external service or executable, they also evaluate much faster than a resource or data source from a provider. If I wanted to build a model of what a basic function looks like, it's going to be something like this. You have the function name and then parentheses, and then some number of arguments to go with that function. Some functions actually take no arguments, while others take many. Arguments are

not named, unlike some other programming and scripting languages, instead, the arguments must be in the proper order. You could test functions by placing them in a Terraform configuration and running a plan, but that's a little time consuming and difficult to debug. For that reason, Terraform has a subcommand called console that opens up an interpretation console where you can have Terraform evaluate functions and other expressions. That is much more efficient than testing things in a configuration directly. Console will also load the current state data values of a configuration, allowing you to use real data to test your functions and expressions. Terraform has over 100 functions, and we are going to cover each one in excruciating detail, so buckle up for 10 hours of excitement. Just kidding. Instead, why don't we group the functions into some broad categories that will help you find the function you need in the future.

Common Function Categories

Based on the current Terraform documentation, there are at least nine function categories with more possibly coming in the future. I'm not going to list all of those categories here. Instead, we'll focus on those you'll probably use as you build your first configurations. The first category is numeric functions. These are functions that are used to manipulate numbers. For instance, if I had a list of numbers and I want to get the smallest number, I can use the min function, and it will return, in this case, the number 7. There are also string manipulation functions. A possible use for a string function is working with Azure Storage account names. They cannot have uppercase letters, and if someone provided a string with uppercase letters for a storage account, you would receive an error. There is a function called lower that takes a string and will put anything that's capitalized into lowercase and return that string to you. There are functions to deal with collections. And when I'm talking about collections, I'm talking about lists and maps, basically. We will be using the merge function shortly in our configuration to merge the common tags map with another map. One interesting category is the functions for IP networking. If you've done any work with IP networking, you know that math in IP addressing is kind of funky, and for that reason, the standard numeric functions don't work very well. There are dedicated functions like cidersubnet(), which takes a network range, carves out a subnetwork in that range based off of arguments that you give it. There are also functions that interact with the local filesystem. One of the most common ones to use is the file function. That takes a path argument pointing to a file, reads the contents of that file out to a string, and returns the string. So if you need to get the contents of a file, you use the file function, very straightforward. Lastly, type conversion functions allow you to convert one data type to another. You probably won't use most of these functions often with the exception of toset(). If you'll recall from earlier, the for each argument takes a map or a set, and the toset() converts a list or a tuple to a set. That's pretty useful. If you're interested in looking at the other categories and functions, they are all nicely laid out in the Terraform documentation. Let's take a deeper look at some individual Terraform functions we will use in our configuration.

Function for the Configuration

We can leverage the functions in Terraform to meet the requirements given to us by John. We'll start with the Startup script. Currently we define the Startup script using a

heredoc expression, but John wants us to move that into a file. We could try using the file function, but we need to dynamically update the bucket name used by our script. Instead, we will use the templatefile function, which reads in the contents of a file and replaces variables in the file with values submitted as part of the function. Another request from John was to simplify the networking by determining the subnet addressing dynamically. We can do that by leveraging the cidrsubnet function, giving it the VPC cidr_range and carving out space for our subnets. The function takes the cidr_range you would like to work with, the subnet bits to add to the existing subnet mask, and which network number you want out of the resulting subnetworks. John also wants us to consistently name all of the resources. We can do that by adding a variable for a naming prefix and adding a name tag for each resource that doesn't have a name argument. But we already have a list of common tags in a map. What are we going to do? No problem. We can use merge to merge our common tags map with a map of additional tags for each resource. Finally, we are going to be adding a naming prefix variable to the configuration. When we use it for our bucket name, we should apply the lower function to make sure our bucket name is always lower case, even if someone submits an uppercase value with the naming prefix. Ready to add some functions to the configuration? Awesome, let's go.

Testing Functions with Terraform Console

Before we try to add functions to our configuration, let's first test out some of these functions using the Terraform console. I'll go ahead and open up the commands directory and open up m7_commands, which has some examples for us to run to try the different functions and syntax. You do need to initialize the configuration before terraform console will work. We've already initialized our configuration, so we don't have to worry about that. We can simply run terraform console. This starts the interactive environment where we can test different functions, and we can make use of variable values and resource and data source values within our functions. Let's first test out a basic numeric function, the min with the arguments 4, 5, and 16. I'll go ahead and copy this and paste it down below. The result is 5, which is correct. That is the lowest number. Now let's try using the lower function that takes a string. TACOCAT, in all capitals, should evaluate to tacocat in all lowercase. I'll go ahead and copy that one, and paste it down below, and there we go. It evaluated our string and set it to all lowercase. Now we're going to test out the cidrsubnet, and we're going to feed it the vpc_cidr_block as a value. The value it uses for the variable is the default value we defined for that variable. If you don't remember, let's go ahead and open up the variables file, and let's find that vpc_cidr_block. It's set to 10.0.0.0/16. Based off of the syntax, we are going to add 8 bits to that to make it a /24. And the 0 argument says we're going to select the first available network from the set of subnetworks. So let's go ahead and run this command and see what the resulting value is. The resulting value is 10.0.0.0/24. That's exactly what we would expect. And this is an excellent way we can leverage the cidrsubnet function to automatically generate the cidersubnet ranges for the subnets we're creating in the count loop. Before we get to that, let's try a few other functions. The next one I want to try is lookup. The lookup function is used to look up the value in a map. You first have to specify a map in the argument. We'll use local.common_tags, and then the key that you want to look for. We'll specify company. If that key is not found, we can give it an alternate value to return, in this case, unknown. Let's try out this function now. And it returns the value Globomantics based off the key, company. If we instead use a key of missing, which I

know is not in the common_tags map, then it returns the alternate value we specified, Unknown. In addition to trying out functions, you can also just retrieve a value. Let's retrieve the value stored in local.common_tags, and it returns the map that's stored in our current local.common_tags. You can also try out some arithmetic operators. One arithmetic expression we're going to use is the modulo operator to assign an instance to a subnet. Now that might not make sense right now. We'll get into that in a moment. Let's first start by moving our startup script to a separate file and making use of that template file function.

Using the Templatefile Function

We are going to move our startup script to its own file and make use of the templatefile function. So for now I will hide the terminal, and let's create a new file in globo_web_app that's going to hold our startup script. I'll name that file startup_script.tpl. You don't have to name it .tpl, that's something that I do just so I know it's a template file. Within that template file we're going to have our startup script. So let's open up the instances file and we'll copy the script that we've defined in user data. So I'll go ahead and copy this entire script and paste it into the startup_script file. Now you'll note in the script we are referencing the aws_s3_bucket.web_bucket.id attribute. When the template file is evaluated, it's not going to be able to directly evaluate that expression. We need to put a variable in here that we can reference in our template file function. So let's change this instead to s3_bucket_name. That's a variable we can now reference in our template file function, and I'll update that for the second entry as well. Now I'll go ahead and save the file, and back in instances we will replace the current user data arguments with the templatefile function. So I'll go ahead and delete what's in here and start this off with a templatefile function. The first argument in the templatefile function is the path to the template file we want to use. To start off the path to the startup script, we can make use of a special variable that exists in Terraform, it's the path.module variable. This will resolve to the full path of the module that we're currently working in. Then we can add a slash and the startup_script.tpl. That's the path to the file we want to use, and now we can provide a map of variables and values to use in that template file. So I'll add a comma, and then I'll start a map with curly braces. We only have one variable in our template file, which is s3_bucket_name, and I'll set the value to aws_s3_bucket.web_bucket.id, which is what we had in the initial startup script. Now it will pass that value and replace wherever it sees s3_bucket_name with that value. Go ahead and save that. The next thing to do is add the CIDR subnet function to our definition of the subnets.

Using the Cidrsubnet Function

We are going to use the cidrsubnet function to get the CIDR ranges for our subnets. Let's go ahead and open up the network file and scroll up to where our subnets are. There we go. There's the cidr_block argument. My challenge to you is to use the cidrsubnet function to define the value for the cidr_block for our AWS subnets. Go ahead and try that now, and when we come back, I'll show you my solution for setting that cidr_block argument. All right, let's see how you did. We're basically mimicking what we did in the console. So we have cidrsubnet the function, we're passing it the cidr_block we're using for our VPC, we're adding 8 bits to the subnet mask, and we're using count.index to select the subnetwork that's

evaluated by adding those bits. As we saw at the console, the first one should evaluate to 10.0.0.0/24, and the next one will evaluate to 10.0.1.0/24. While we're still thinking about networking, there's something we need to update about our instances. Let's go over to the instances.tf file. You'll notice for the subnet_id we're using the expression aws_subnets.subnets[count.index].id. That's going to work well when we have two instances and two subnets, but what happens if we want four instances distributed evenly across two subnets? This expression is not going to work anymore because count.index will go beyond the number of subnets that we have. We need a different expression here that evaluates properly, and we can use the modulo expression to do that. So let me show you how that works by bringing up the console again. Going with our example, let's say that we have four instances, and we want to place it in either the first or second subnet. We can use the modulo operator to do that. We would start with the count.index of the first instance, which would be 0, and then the modulo operator, which gets the remainder after a division. And then we'll look at the number of subnets we have, which is 2. That will evaluate to the number 0, so it will get placed in the first subnet. Our next instance will be instance number 1, and when we do % 2 on that, we'll get a remainder of 1. So far, so good. That's going to go in the second subnet. Now, our third instance, if we do % 2 on that, because there is no remainder, that will resolve to 0, and it will put it in the first subnet. Our fourth instance will evaluate to 1, so it will be placed in the second subnet. All we need to do is encapsulate this modulo expression so that it puts our instances evenly across two subnets, and this will work for more than just two subnets. Let's go ahead and update our expression to be [(count.index % var.vpc_subnet_count)], and we'll put the whole expression in parentheses so it's evaluated before it tries to get the element. Now we can increase the number of instances beyond 2 and not worry about how they're distributed across our subnets. I'll go ahead and save this file now. The next request to deal with from John is consistent naming across all of the resources, so let's get started on that.

Adding a Naming Prefix

John wants a naming prefix to be added to the configuration and then consistent naming across all resources. Let's go ahead and start by adding a variable to the variables file. We'll scroll down and open up the variables file. There we go. We'll create a new variable and name it naming_prefix. We'll set the type = string, the description = Naming prefix for all resources, and we'll set a default of globweb. Instead of using this naming prefix directly, why don't we add a local value where we manipulate this naming prefix a little bit? So let's go ahead and open up the locals.tf file, and we'll add a new local value here called name_prefix, and we'll set that equal to the variable naming_prefix and add -dev to it. We could update this for each environment as we create them, and that's something we'll deal with in a later module. The S3 bucket name does not currently use the name_prefix. So my challenge to you is to use this new local value name_prefix and add it to the S3 bucket name instead of globo-web-app, and then make sure that the entire S3 bucket name is all in lowercase. Go ahead and try that out now. And when we come back, we'll see my solution. Alright, here's how I approach that solution. I'm using the lower function here to make sure everything in the string is set to lowercase. I'm referencing the name_prefix local. Did you know you can reference a local inside of ae locals block? You sure can. I'm referencing the name_prefix local, dash, and then the random integer result for the bucket name. The last thing to do is add common naming to all the resources within our configuration. Let's first start with the VPC. I'll

go ahead and open up the network.tf file, and let's scroll up to where we define our VPC. We can add a name tag for our VPC by adding an additional tag, but we already have our local.common_tags. How are we going to combine that with a new tag? We can use the merge function. Let me show you how the merge function works. Be sure to save both the variables and locals file before you try this expression. I'll go ahead and bring up the terminal again so we can test out the merge function in the terraform console. I exited out of the terraform console because it didn't have our new variables and locals loaded into it yet. It only loads those values when you first start up the console. To make sure I have the latest variables and locals, I am going to launch terraform console again. Now we can try to use the merge function. We're going to try to merge together the local.common_tags and add a new map that has a name tag in it. We start with the merge function, and then in parentheses, the first map we want to use is (local.common_tags, and then we want to add another map to it, so we'll add a comma and start another map with the curly braces. Within that map, all we want to add is a name tag. So we'll start with the key, Name, and set it equal to the local.name_prefix value and add on -vpc for the string. Then we'll close our map with the curly braces and close our merge function with parentheses. And if I hit Enter, there we go. We have an updated map that will be submitted to the tags argument that has all the values we want. So I'll go ahead and close the terminal, and now we'll update the tags with the merge function. I'll add the merge function, add a comma after local tags and start a map. We'll set the key to Name. We'll set the value to the same value we just used in the console, local.name_prefix and add -vpc. And then we'll close the map and close the parentheses for the merge function. There we go. That's all done. Now my challenge to you is to go through the rest of the resources within the configuration. If there's a name argument, go ahead and update the name argument as needed. If there is no name argument, add a name tag to the tags argument using the merge function. A good example of a resource that does have a name argument, if we scroll down to our first security group, this does have a name argument, which will be applied as the name tag. So we can update this one with the naming prefix. Any resources that don't have this name argument, you can add it to the tags argument instead. One other thing to note is for resources where we're using the count argument or the for each argument, you may want to use that value in the naming tag, so you name each instance of the resource differently. Subnets would be a good example of that where you want to name each subnet based off of the count index. Go ahead and try to make those changes now. When we come back, we'll go through the process of formatting, validating, and deploying our updated configuration.

Validating and Deploying the Updated Configuration

Now that you've updated your configuration files, if you have any questions, you can definitely check out the m7_solution directory to see my solution for this configuration. Let's go ahead and format and validate our configurations. I'll exit the Terraform console by doing exit, and go ahead and run terraform fmt. All of our files are now nicely formatted, let's run terraform validate to make sure we don't have any syntax issues, and it looks like we missed something with our subnets, we're going to have to update our configuration. Let's go into the loadbalancer.tf file, and scroll all the way up to our load balancer, and sure enough, we're referencing two resources that don't exist anymore. We need to update this. What we want is a list of all the subnets that exist. Now if you'll remember from the reference syntax we looked at earlier, we can do this by using an asterisk, so I'll change the reference to

aws_subnet.subnets, and then we'll use square brackets and the asterisk to indicate that we want to retrieve all of the instances, and then we'll use .id to get the ID attribute of all of those instances. Now the object returned is going to be a list, so we can get rid of this other reference of subnet2, and we can get rid of the square brackets, because the object being returned is already a list, we don't want to put a list inside of a list. So I'll go ahead and save this, and now, let's try running terraform validate again. And success! Our configuration is valid. That's why you always run terraform validate after you make changes, because you're always going to forget something in your configuration. Okay, with a valid configuration, we can go ahead and run through the plan and apply process. Going back to m7_commands, if you haven't already exported your AWS access key and secret key as environment variables, go ahead and do that now. I already have that set, I can move onto the next step, which is running terraform plan, and sending the plan to the file m7.tfplan. Go ahead and copy that, and run it down below. This is probably going to make some significant changes, because we're changing the naming, we're changing some of the subnet references, it's going to make a lot of changes is what I'm saying. So, it's going to add 19 new things, change 3, and destroy 19 things; and that's because, like I said, we're making some pretty significant changes. Fortunately we're still in a development context, so we can go ahead and run terraform apply to apply these changes to our deployment. Since we're recreating a lot of stuff, this is going to take a while to apply, so I'll go ahead and pause the recording now, and we'll resume when the deployment has completed. Okay, our deployment has completed successfully, it's made all the changes that we asked of it, let's go ahead and make sure our application is back up and running. I'll go ahead and copy this entire address right here, and we'll go over to a browser, and paste it in here. There we go, our site has loaded successfully. Let's go over to the EC2 Management Console. Looking at the names for our EC2 instances, we've got globoweb-dev-nginx-0 nginx-1. I used the count index to help with the naming of the instances. If you'd like to go to your AWS console, you can verify the naming and the creation of all of your resources to make sure your configuration operated correctly. At this point, we have satisfied all of John's requirements. Well done everybody.

Summary

In this module, we explored the concepts of looping and functions in Terraform. We started with adding count and for each loops to make our configuration more dynamic. Then we leveraged functions to further improve the code and simplify the required inputs for deployment. In the next module, we are going to examine modules. Oh boy, that might sound a little confusing. We'll examine Terraform modules. Terraform modules are used to package up common configurations for reuse, and they are incredibly useful. That's coming up, next.

Using a Module for Common Configurations

Overview

A common feature of programming languages is the ability to import libraries or modules for common tasks, data structures, or functions. Terraform implements a similar ability through the use of modules. Terraform modules stop you from reinventing the wheel by allowing you

to use common configurations built by others. Hey everyone, this is Ned Bellavance, I'm a HashiCorp Ambassador and founder of netinthecloud. Let's dive into Using a Module for Common Configurations. We'll start this course module by first defining what a Terraform module is. The most surprising thing? You've been using a module this whole time and didn't even realize it. Once we've established what a module is and how it's used, we'll check in with John to see what improvements he thinks we could make by leveraging modules in our code. Then we'll implement those changes, first by using an existing module from the Terraform public registry, and then creating our own module for S3 buckets. But first, what is a Terraform module?

Terraform Modules

Whether or not you realize that you've been using Terraform modules all along, what is a Terraform module? It is simply a configuration that defines inputs, resources, and outputs, and all of those are optional. When you create a set of tf or tf.json files in a directory, that is a module. The main configuration you are working with is known as the root module, and you can invoke other modules to create resources. Modules can form a hierarchy with the root module at the top. Our root module could invoke a child module, which could in turn invoke another child module. For instance, let's say we use a module to create a load balancer with a VPC and an EC2 instance. The load balancer module may use a module to create the VPC and another to create each EC2 instance. The motivation behind creating or using modules is to leverage a common set of resources and configurations for your deployment. Where can you get modules? They can be sourced from the local filesystem, a remote registry or any properly implemented website that follows the HashiCorp provider protocol. The most common source is the Terraform public registry. In fact, you may have noticed the browse module option on the public registry. Modules that are hosted on a registry are also versioned in the same way that providers are. You can specify a version to use when invoking a module. Staying on your preferred version can prevent breaking changes from impacting your deployments. Once you've added the module to your configuration, terraform init will download the module from the source location to your working directory. If the module's already in the current working directory, Terraform will not make a copy of it. As I mentioned in the discussion of looping, you can create multiple instances of a module using either the count or for_each meta arguments. The components that make up a module should already be very familiar to you. Modules generally have input variables, so you can provide values for input to the module, and output values that are based off of what the module is creating, and, of course, the actual resources and data sources within the module. A module is not required to have any of these components, but it probably would not be very useful without them. Now that we know a bit about modules, let's see what Globomantics has in mind for using them with our configuration.

Globomantics Updates

After learning about modules, you're probably already thinking of how the configuration could be improved and simplified. Looking at the current architecture, we've got our application sitting in a VPC, which we know is made up of subnets, routes, route associations, an internet gateway, and more. That seems like a common configuration that should go into a

module. Also, our S3 bucket with the bucket and the IAM policies seems like something that might be used in other deployments at Globomantics. What does John think about that? You caught up to John in the hallway to talk about your module idea. Turns out, he's been researching them on his own, and he thinks it would be a great idea to add them to your configuration. He would like everybody at Globomantics to standardize on the VPC module from the Terraform public registry. He also really likes the idea of creating a module for the S3 portion of the configuration, with all the necessary IAM roles, profiles, and bucket policies that will allow a load balancer to write access logs and EC2 instances to grab website content. Before we try to implement these improvements, first, we need to check out the syntax used to create and instantiate a module.

Module Structure

A module really is just a collection of Terraform files in a directory. In the same way that we have been crafting our configuration with files for variables, outputs, and different resource groupings, you can do the same with a module or just put it all in one big file. The contents of a module will include input variables, resources to be created, and outputs that the parent module might want to use. In this example, we are building a module for an S3 bucket. The input variable bucket_name can be used to name the S3 bucket. Then we have the actual resource being created. And in that resource, we can use the input variable bucket_name. The parent module is probably going to want some information about that bucket, and we can expose that information using an output value, passing back the bucket_id. Now is probably a good time to mention why input variables and output values are so important to a module. The only way for a parent module to pass information to a child module is through input variables. The child module has no access to local values, resource attributes, or input variables of the parent module. Any information a module might need has to be passed through those input variables. Likewise, the parent module has no access to the local values and resource attributes of the child module, the only way to pass information back to the parent module is through output values. The good news is that the input variables and output values support any data type available in Terraform, so you can pass a complex object, an entire resource, or a simple string. The choice is up to you. Bearing this in mind, let's take a look at how you instantiate a module, pass variable values to it, and reference outputs from it.

Module Syntax

Invoking a module starts with the module keyword, followed by a name label for the module. The rest of the configuration information goes inside the block. The source argument tells Terraform where to get the module from. The source can be the local file system, the Terraform registry, or any other supported source type. If the source supports versioning, you can specify the version argument with a version expression, just like the expressions we used for a provider version. If you'd like to use a specific instance of a provider within a module, you can do so with the providers argument. The value for the argument will be a map where the key is the name of the provider in the child module, and the value is the name of the provider alias in the parent module. If you don't specify a providers block, Terraform will use the default provider instance. The remainder of the block will be key-value pairs, with the

key being the name of an input variable in the child module and the value that you would like to pass to the child module. Let's take a look at an example with our potential S3 bucket module. We'll start with the module keyword and a name label of taco_bucket. In the block we'll specify the source as the current directory and the S3 subdirectory where we have our module files. Since this is a local source, it doesn't support versions. Beyond that, we can pass the single variable bucket_name to the module, and that's it. The module will take care of creating the resources and making output values available. Let's take a look at how we can reference those values. The general format for referencing a module output is the module keyword, dot, the name label of the module, dot, the output value name. In our example, we can get the bucket id by using the syntax module.taco_bucket.bucket_id. As I mentioned, the output value can be any data type, and naming is up to you. We used the attribute name in this example, but we could've called the output value s3_bucket_id or whatever else makes sense in the context of the module. Armed with all of this module knowledge, let's start making use of it. First, we will replace the current VPC resources with the VPC module on the Terraform registry. Of course, that means going to the registry and reading about the VPC module.

Adding the VPC Module

Here is the main page of the Terraform Registry. We've already gone into the Providers section, so now let's go into the Modules section. If we look to the left, we can filter by Provider for the modules that we want. The module we actually want is the vpc module, which happens to be the top module here. So let's go ahead and click on that module. Within this module, I want to point out a few things. It has a basic set of instructions on how to use the module. It also provides the source code for the module on GitHub. So if you want to view what's actually in the module, you can click through on that link and view it yourself. Scrolling down a little bit, we have a README section, which describes how to potentially use this module. And then it also gives us a list of inputs that are accepted by the module, output values that are given by the module, any dependencies and the resources that would be created by the module. A lot of this is dependent on the inputs you give the module. We're going to be setting up a fairly basic VPC, so we can simply copy this example and paste it into our configuration and then make some simple updates. Let's do that now. Okay, I've copied the text, let's head over to our configuration, and I'll expand globo_web_app and open up the network.tf file. There we go. And, we are going to be replacing a bunch of resources with this. We're going to be replacing the vpc, the internet_gateway, the subnets, the route_table, and the route_table_associations, all with this module. That's a lot of resources that we no longer have to manage. Let's scroll back up to the top and paste in the module example. Since this module is from the Terraform Registry, we should definitely add a version argument to pin it to a specific version. This way, if the module is updated in a way that breaks our configuration, we can test it in a development environment before upgrading to the newest version of the module. If we go back to the browser, we can see the current version is 3.10. So let's go ahead and pin it to 3.10 for now. We'll set the version = to 3.10.0, and this way it will only use that version until we change this argument. Okay, now we need to update some of the values that are used for the arguments here. We'll first delete the name argument since we'll be submitting that through our tags. Next, we'll update the cidr block to use our variable. For the availability zones argument, we want to give it a list of availability zone names that's equal to the number of subnets that we're currently using. To

do that, we can use the slice function to slice off a list of names from the availability zones data source. Let's see how we do that. The slice function takes a list as input and then slices off a portion of that list for use. We'll specify the data source aws_availability_zones.names. The next argument is the starting index for the slice. We'll start at the first element in the names list. The last argument is the ending index of our slice, and it is not inclusive, so it won't include that element in our list. We'll set that to var.vpc_subnet_count. So when our subnet count is 2, it will return 2 availability zone names and it will already be in a list format. Okay, for the private_subnets, we don't have any private subnets, so we can go ahead and delete the private_subnets. For the public_subnets, we are going to need to calculate a list of public subnet CIDR ranges for our public subnets. Let's look at how we've done this already. If we scroll down to our subnets, we compute the CIDR block using the cidrsubnet function and the count.index since we're creating our subnets and accounts. We're no longer generating our subnets in account, so we need an alternate way to generate this list of CIDR subnets. The way that we'll do that is with a for expression. I briefly mentioned for expressions in the previous module, now it's time to dig into those a little more deeply.

For Expressions

For expressions are a way to create a new collection based off of another collection object. It's especially useful when you're dealing with resources that have a count or a for each argument. The input in a for expression can be any collection data type, list, set, tuple, map, or even object. The contents of the collection will be available for transformation in the for expression. The result of the for expression will be either a tuple or an object data type. Remember that these are structural data types, which means the values inside don't all have to be of the same data type. To help customize the result, you can filter it with an if statement. You can filter on any value from the inputs. Let's check out the syntax in a for expression to lend some clarity. First, let's see how you would create a tuple result with a for expression. The expression starts with either curly braces or square brackets. The square brackets indicate that the result will be a tuple. I found that kind of confusing at first, so I'll repeat that. The brackets or braces that you use to encapsulate the for expression determine the result type. After the square brackets, the expression starts with the keyword for, followed by syntax that identifies the input value and the iterator term to use during evaluation. The structure is the iterator term, followed by in, and then the input value. After that we have a colon, which signals the start of the value which will be stored in each tuple element in the resulting collection. If that sounds esoteric and difficult to parse, I agree, and I think an example will clear things up. Let's say we have a local value called toppings of type list with three elements in it, and we'd like to create a new tuple with the word Globo added to each element in the list. We can accomplish this with a for expression. The square bracket says that we want a tuple as the result. The t is the placeholder for each value in the input value local.toppings. After the colon is the resultant value we want to use for each element, which is simply the string Globo and the value stored in placeholder t. The resultant tuple will have three elements, Globo cheese, Globo lettuce, and Globo salsa. Remember that the input value doesn't have to match the result. Local.toppings could have been a map. Now let's check out the syntax for creating an object. The expression will start with curly braces to indicate that we want an object as a result. As a quick refresher, an object is basically a set of key-value pairs where the values can be of different data types. In this expression, the input value is a

map, which means we now need two iterator identifiers, one for the key and the other for the value. Next, we have a colon and an expression to evaluate for each entry in the object. The syntax is the object key, followed by equals and the greater-than symbol and then the object value. Again, an example will probably help. Here's a local value called prices of type map with three key value pairs. Let's say we'd like a new object where each price is rounded up to the next whole integer. We can do that with a for expression, where i is the map key, and p is the map value. The expression to evaluate keeps the same key i, but alters the value with the ceiling function. The resulting object has the value for each pair rounded up to the closest integer. We are going to use a for expression to dynamically generate a list of subnet CIDR ranges. Let's head over to the configuration and see how.

Using a For Expression

We are trying to create a tuple of cidrsubnet addresses to pass to the public subnets argument. The number of elements in the tuple should equal the value stored in vpc_subnet_count, which means we'll need an input to the for expression that is a list of integers from 0 to the value in vpc_subnet_count. Fortunately, there is a function called range that will do exactly that. Let me pull up the terminal, and we'll start up terraform console to test out these expressions. First we will test the range function. The syntax is the range function and then the value you want it to count up to. We'll specify the variable vpc_subnet_count. Range will hand back a list of 0 and 1. That sounds good; that's a good start. Now we have a list to use as an input value for the for expression. For the evaluation portion of the expression, we can use the cidrsubnet function, passing it the vpc_cidr_block variable, 8 bits to add to the subnet mask, and the element value in our input list. Let's try to construct a for expression with that information. We'll start the expression with square brackets because we want a tuple back, we'll have the for keyword to start our for expression, we can set an iterator for our list, we'll call it subnet, and that's going to be in the range, and we'll set the range to var.vpc_subnet_count, which we know will return a list with two elements, 0 and 1. Then we'll add the colon and then the expression we want to use for the result of each element. We'll do the cidrsubnet function, we'll feed it the variable vpc_cidr_block, 8, to add 8 bits to the subnet mask, and then the subnet iterator. That will be 0 on the first evaluation and 1 on the second. We'll close that parentheses for the cidrsubnet function and close the for expression with a square bracket. I'll go ahead and hit Enter, and we get back a tuple that has two subnets in it. Perfect, that's exactly what we need. And this will be dynamic based off of the values of the cidr_block and the subnet_count. Let's go ahead and copy this entire expression, and I'll hide the terminal. Scrolling up to our vpc module, we can replace the value for public_subnets with our new expression. Okay, there we go. For enable_nat_gateway, we want to set that to false. We don't have any private subnets, so we do not need a NAT gateway. We'll also set enable_vpn_gateway to false, because we are not using a VPN gateway. For our tags, we can go down and grab the tags argument from our existing VPC resource, and go ahead and paste that as the value for the tags argument in the module. There we go. With our vpc module ready to go, we can remove the other resources. There we go, I have removed all the resources that we're replacing with the vpc module. Now the next thing we need to do is replace any of the references to our VPC resources with the module references. There are two output values that you'll need to use to update the rest of the configuration. The first is the vpc_id, and the second is the public_subnets list. For the vpc_id, we'll update the expression to module.vpc.vpc_id. Vpc_id is

the output value from our vpc module. We also need to update anywhere that the subnets are referenced. For example, let's go to the load balancer. In the load balancer we were referencing all of the public subnets with this expression. We need to update the value for the argument to use the public subnets that are created by the module. To do that, we will update the value to module.vpc.public_subnets, which is a list of all the public subnets. My challenge to you is to go through the rest of the configuration and update it to use these module outputs. If you have any questions, you can always reference the solution that's in the m8_solution directory. Go ahead and pause the video now, and when we come back we are going to create our S3 module.

Creating the S3 Module

The S3 module we are creating should create an S3 bucket with a bucket policy that allows a load balancer to write logs and the proper IAM resources to grant access to an EC2 instance. Our inputs are going to include the bucket_name, the elb_service_account_arn, and the common_tags to be applied to all resources. Those can all go in the file variables.tf. The resources we need to create already exist in our configuration. We've got the S3 bucket itself, the IAM role, the role policy, and the instance profile. In terms of output values, we are going to use the S3 bucket and instance profile. We can simply return the entire object for each resource and make use of all the attributes within. Let's head over to the configuration and start setting up our module. Let's create the S3 module inside of the globo_web_app directory. We'll start by adding a directory called modules, just in case we want to add any future modules to our configuration. Within that modules directory, let's add a subdirectory for our S3 module, and we'll name the directory, globo-web-app-s3, so that's pretty descriptive of what this module is intended for. Within that directory we'll create three files. We'll start by creating the variables.tf, followed by a main.tf file to hold all of our resources, and then an outputs.tf file for our output values. Within the variables file, I'm going to add some comments here for variables we want to create. My challenge to you is to create these three variables, the bucket name, the ELB service account, and the common tags to pass to this module. Go ahead and try that now, and when we come back you can see my updated solution. Okay, here is my solution. I've got a variable named bucket_name that's of type string; a variable called elb_service_account_arn, which is also of type string; and then a variable called common_tags, which is of type map with strings as the values for the map. And I actually set a default for the common_tags in case someone using this module doesn't submit a list of common tags to use in the configuration. Next up, we will add our resources to the main.tf file. So let's scroll down and open up the s3.tf file. We are going to copy all the resources in here except for the S3 bucket objects. Those will still be created in part of our main configuration. First, I will grab the S3 bucket resource, remove that from the s3.tf file, and paste it into the main.tf file. Next, I will grab all of the IAM resources below the bucket object resource and paste those into the main.tf file. Let's go ahead and save the s3.tf file. And now back in the main.tf file, my challenge for you is to update the references here to use the input variables for all of the resources. Go ahead and try that now, and when we come back we can review my updated solution. Okay, in my updated solution the bucket value is going to be var,bucket_name. In the policy statement, we're now using the variable elb_service_account_arn instead of the data source, and for the references to the bucket_name, we'll use the variable bucket_name. Scrolling down to the end of the S3 bucket resource, the tags have been updated to use the var.common tags. For the IAM role, I've

updated the naming to use the bucket_name as the beginning of the name for the role, and I've updated the tags to be var.common_tags. For the role policy, I'm also using the bucket_name to name the role policy and updating the reference to the bucket_name to use the bucket_name variable. And down in the instance profile, I updated the name to use the bucket_name for naming and also updated the tags to use var.common_tags. That's everything that's in the main. Now the last thing to do is to create two outputs. I'll go ahead and save the main.tf file, and let's go over to outputs and I'm going to put two comments in here for the bucket object and the instance profile object. My challenge to you is to add the output values here to pass the whole bucket object and the whole instance profile object back up to the parent module. Go ahead and try that now, and when we come back we can view my updated solution. In my updated solution, we have the output web bucket, which is set to a value of aws_s3_bucket.web_bucket. Because we don't specify an attribute, it will pass the entire bucket object back as an output value. That's pretty useful. And then we do the same thing for instance profile, referring to aws_iam_instance_profile.instance_profile. We'll go ahead and save this output file, and now we need to add the module reference to our s3.tf file. I'll go ahead and select that now. Now my challenge to you is to add the module block with the proper input variables. Go ahead and pause the video now, try it out, and when we come back you can see my updated solution.

Adding the S3 Module

Okay, here is my updated solution. We're creating a module with the name_label web_app_s3. The source will be the current working directory /modules/globo-web-app-s3. We have to give it three input variable values, bucket_name set to the local.s3_bucket_name, elb_service_account set to the elb_service_account data source, and common_tags set to local.common_tags. Now that we've updated to use a module, we're going to have to update any references to the bucket or the instance_profile with the output values from this module. For instance, our bucket argument in the aws_s3_bucket object should be updated to module.web_app_s3.web_bucket.id. My challenge to you now is to update any other references to the bucket or the instance_profile in our configuration to use the proper output value from our S3 module. Go ahead and do that now, and then we'll review my updated configuration when we come back. Okay, we're back in the loadbalancer file. I simply updated the access_logs argument to use the S3 bucket created by our module. Over in instances, the iam_instance_profile has been updated to use the instance_profile output value, .name. We had to update the depends_on, because it was referencing the iam_role, but that's not available to us anymore, so instead we just make it dependent on the module itself. And then in the templatefile function we have to update the s3_bucket_name to be the web_bucket output value and the .id attribute. While we're looking at the instance configuration, one thing I want to point out is the configuration for the subnet_id. At the end of the module expression, I have removed the .id attribute, and that is because what's returned by the module is actually a list of public subnets and not the public_subnets object itself, which would have the id attribute. That's going to do it for all the updates to our configuration. Go ahead and save those updates. The next step is to get those updates deployed.

Validating and Applying the Updated Configuration

We have updated our configuration to use a VPC module from the Terraform Registry and an S3 module that we wrote ourselves. Let's go ahead and get those changes deployed. I'll go ahead and open up the m8_commands file. And the first thing we need to do is run Terraform in it because Terraform needs to get these modules and include them in the configuration. So I'll go ahead and open up the terminal. There we go. And we'll run Terraform in it. Okay, Terraform has successfully initialized. If we scroll up, we can see under initializing modules that it downloaded the VPC module and placed it in the .terraform\modules\vpc folder. Because our globo-web-app-s3 module is already in the local directory, it will not try to download or copy the files for that module. All right, now that we have successfully initialized Terraform, the next step, of course, is to run terraform fmt. But we don't just want to format our files in the globo_web_app directory, we also want to make sure that the files in our S3 module are formatted properly. And we can do that by running terraform fmt-recursive to go into those subdirectories and properly format those files as well. And there we go, it has updated the formatting for all of our files. The next step is to run terraform validate. All right, excellent. Our updated configuration is valid. The next step would be to export your environment variables if you haven't already done so. And after that, we'll run Terraform plan and send the output to m8.tfplan. We'll go ahead and run that now. Since we're moving a lot of things to modules, it's also going to have to recreate a lot of our infrastructure. Again, I'm glad we're doing this all in development before we roll anything out to production. Let's go ahead and run terraform apply to apply all of these changes. If you happen to be running in production and you needed to move resources from the root module to a child module, that's a case where using the terraform state mv command can help you move things from the existing address to a new address that's inside the module, and that would stop Terraform from destroying the target infrastructure that you're trying to update. That's a pretty advanced topic, which we're not going to get into here. For our purposes, we're still in the development environment, so tearing down this infrastructure and recreating it is no big deal. This is going to take a while, so I'll go ahead and pause the recording now, and we'll resume when the deployment has completed successfully. All right, our deployment is successful, let's go and check on our website. Copy that address, and go over to a browser. And there we go, our website is up and running and we are now using a VPC module and an S3 module. That's awesome.

Summary

Terraform modules are incredibly useful. You can find existing modules on the Terraform Registry and save time and effort by not reinventing the wheel. You can also write your own modules to assist with creating common configurations in your organization and share those modules internally or publish them on the Terraform Registry. It turns out we've been using modules this whole time. The root module is simply the module you're currently working in. It has input variables, resource and data sources, and output values, just like any other module. The last thing we will cover is how to use the same configuration to spin up multiple deployments. We can leverage Terraform workspaces to manage state data for multiple environments. That's coming up in the next module.

Using Workspaces for Multiple Environments

Overview

One of the great things about infrastructure as code is its reusability. With a few minor tweaks, the same code can be used to deploy nearly identical infrastructure in multiple environments. Terraform has a special feature called workspaces to help with reusability. Hi everyone, this is Ned Bellavance. I'm a HashiCorp Ambassador and founder of Ned In The Cloud. Let's dig into supporting multiple environments with Terraform We'll start the module by talking to Sally Sue over in software development. She's ready to move this web project out of development and into production, and she wants to make sure each environment is consistent. With that objective in mind, we'll take a look at how Terraform can be used to handle multiple environments with the same configuration. We will have to consider things like input values and state data management. One way to handle state data is with Terraform workspaces, and we will put that into practice with our configuration. But first, let's have a chat with Sally Sue

Globomantics Expansion

Our configuration and deployment for the web app at Globomantics has evolved and improved since that base configuration was handed to us. Now Globomantics is ready to roll this little project into a staged environment workflow. We can think of our current deployment as the development environment. We've kept things small with tiny EC2 instances and only using 2 subnets. That is probably not going to work so well in a production environment. Sally Sue has approached us about adding a user acceptance testing environment and a production environment to the existing development environment. She would like us to make sure we are using the same configuration for each environment but supplying different input values depending on the environment. Fortunately, we have used variables extensively in our configuration, so it should be relatively easy to make adjustments for input values. Then we can feed those input values into our single configuration and use it to create and maintain each environment. But what about state data, credentials, and the deployment workflow? Let's examine what it means to deploy multiple environments and how to leverage Terraform workspaces.

Terraform Workspaces

We are going to support multiple environments from a single configuration. Before I introduce Terraform workspaces to help with this goal, let's first think about some of the challenges inherent in supporting multiple environments. When you're working with multiple environments in Terraform, there's a few things to bear in mind. First of all, generally speaking, your environments are going to have more in common than they have differences between them. That's kind of the whole point of having multiple environments in a configuration. Your dev should be very close to your UAT, and your UAT should be very close to your production because anything you test and validate in UAT should be what actually ends up in production. It also means that it's useful to have abstractions within your configuration where you can apply those different values, making your code more

reusable. We've done that by making extensive use of input variables. Another thing to consider in the whole process is access. You're probably not going to have access to production if you're the one deploying to the lower environments. Often there's a separation of responsibility and access between what's called the lower environments and the production environment, so it's important to keep that in mind, especially when you're thinking about dealing with access keys and secret keys. One of the ways to create multiple environments in Terraform is by using workspaces, and this is the HashiCorp recommended way of working with multiple environments. We'll see how that works in a moment. There are some decisions you need to make when it comes to having multiple environments. First is the management of your state. Where is your state data going to live and how are you going to manage the state data for multiple environments? Typically, it's not a single state file for all of your environments. Instead, you'll have your state data stored separately for each environment, or in more complicated setups, you might actually have a state file for your networking and a separate state file for each application running in an environment. That's a lot of state data to manage. Then you have to determine where you're going to store your variable values. Where are these values coming from? Are you going to store them in a file, are you going to submit them at the command line or are you going to use some third-party tool to generate these values and submit them to Terraform? You also have to think about credentials management. Like I said before, you're not necessarily going to use the same credentials to deploy to production as you do to the lower environments. And in fact, in a lot of places, you use a different set of credentials for each environment. How do you manage those credentials and where are they stored? And finally, there's a balance to be struck between the complexity of your configuration and the amount of administrative overhead there is to maintain that configuration. You could go with something that is relatively simple, but requires a significant amount of admin overhead or make something that's fairly complex, but also dynamic and robust, so when you want to add or edit an environment, there's not a whole lot of administrative work to do. Let's look at two examples of how you could potentially manage multiple environments.

State Management

In this example we are going to manage our environment using multiple state files and multiple configuration value files. We have our primary folder where our Terraform configuration lives, along with a common set of variables that is the same across all environments. And then we can have folders for each environment, dev, uat, and prod. When we are running our terraform plan, we can specify that we want to store the state file in one of those directories. For instance, if we're running terraform plan for development, we can say, place the state file in the dev folder and call it dev.state. Then we can specify a variable file called common.tfvars that has our common values within it. And then finally, an additional var file called dev.tfvars that's in the development folder that has our development values. Now everything to do with development is stored in the development folder. We could proceed with the same for uat and with production. That's one way you can manage your state data and your variable values. Another potential way is to use workspaces. In workspaces, you still have your primary directory where your main configuration and your Terraform tfvars files exist. Workspaces will manage the state for you. It creates a terraform.tfstate.d directory and places the state files and child directories within that main directory. When you want to create a new workspace, you simply run the command terraform

workspace new, and the name of the workspace. Terraform will create that workspace and switch you over to that context, and then you can just run terraform plan using your main configuration and the terraform.tfvars. Rather than manually managing your state, now Terraform is managing the state for you. But how do you get the individual value settings for each environment dynamically based off a workspace? Let's take a look at how we could do that in our configuration.

Adding Workspaces to the Configuration

For our three environments from Sally Sue, she would like us to change the following values based on environment, the VPC CIDR range, the subnet count, the instance type, and the number of instances. She wants us to use the values that you see in this table. We can accomplish this goal by using a map for each variable and the special value terraform.workspace, which resolves to the currently selected workspace. Let me show you what I mean by making an update to our configuration. As I just mentioned, there is a special value called terraform.workspace, which evaluates to the currently selected Terraform workspace. Let's bring up the terminal now, and I'll go ahead and enter the terraform console. We can retrieve the special value by simply typing in terraform.workspace. The current terraform.workspace value is default, and that's the only workspace we have available. The default workspace cannot be deleted, and it's selected by default When you create a new Terraform configuration. we can make use of the terraform.workspace value throughout our configuration. For example, we could update a setting in our locals.tf file. Let's go ahead and expand the globo_web_app directory and open up the locals.tf file. I'll go ahead and hide the terminal to give us some more room, and if you remember from earlier, we created a name_prefix local value, and we added -dev to the end of that value. But instead of doing that, why don't we use the name of the terraform.workspace? So I'll go ahead and delete dev off of the end and update the value to terraform.workspace. Now the naming prefix will reflect the environment that it's deployed in. We can also add an additional common tag called environment and set that equal to terraform.workspace. Now any resource that uses the common tags will have an environment tag equal to the terraform.workspace. We can also use the value of terraform.workspace to select a value from a map in our variables. So let's open up our variables file and update the type for one of the variables that Sally Sue specified. As a quick reminder, those variables are the CIDR block, subnet count, instance type, and instance count. Let's update the variable vpc_cidr_block with the three CIDR block values she wants for development, uat, and production. I'll update the type to a map of strings, and I will remove the default value and make sure that we specify an appropriate map value in our terraform.tfvars file. Speaking of which, let's go ahead and open that up. In our terraform.tfvars file, we will add a value for that variable. So let's go into split-screen mode here. We'll have terraform.tfvars open on the right and variables.tf open on the left. I'll grab the variable name from the left side and paste it in here, and it has to be set to a map. And we'll add three map values in here, one for development, one for uat, and one for production. We'll set Development to 10.0.0.0/16, UAT to 10.1.0.0/16, and Production to 10.2.0.0/16, just like Sally wanted us to. Now how do we make use of this in our configuration? I'll go ahead and save the terraform.tfvars file and exit out of split-screen mode, and let's open up the network.tf file. Within the arguments for our VPC module, we now have to update the way that we are getting a value out of our variable vpc_cidr_block. And since it's now a map, all we have to do is add square brackets at the end

and set the value we want to retrieve to terraform.workspace. Now Terraform will evaluate what workspace it's currently in and select that value from the map that we have stored in vpc_cidr_block. That means we have to make sure when we create our workspaces, we name them to match the keys that are in the map for the vpc_cidr_block. My challenge to you is to update the other three variables and add values into terraform.tfvars and update all the variable references in the configuration to use terraform.workspace. Once again for your reference, here are the values to use for each environment. Go ahead and try to make those changes now, and when we come back we'll take a look at my updated configuration. Okay, let's see how you did. In the variables file, the vpc_subnet_count should now be a map of numbers, the instance_type should be a map of strings, and the instance_count should be a map of numbers, and all of them should have no default value. In our terraform.tfvars file, you'll now have an entry for each of those variables with the values set for each environment. Over in our network, just to take a look at how the vpc module is configured, we already updated the CIDR block, for the availability zones we had to add the terraform.workspace to the vpc_subnet_count, and then also add it for the public subnets for the vpc_subnet_count and the vpc_cidr_block. The easiest way to update everywhere is to do a simple find and replace of all the instances of those variables with the appended square brackets and terraform.workspace. All right, now that our configuration is updated, it's time to make use of Terraform workspaces. But before we do that, we have to talk a little bit about dealing with sensitive data.

Managing Sensitive Data

Credentials and other sensitive data is going to be part of your Terraform configuration. The question is how to deal with that information and keep it secure. One option is to store it in a variables file that is not committed to source control. That is not especially secure, but it sure is easy. The option we've selected for our AWS credentials is to store them in environment variables, and you can use that for any variable in your configuration. It's not uncommon in a deployment pipeline to load sensitive values from a secrets management service into environment variables on the system running the deployment. That is definitely more secure. You want to make sure to mark those variables as sensitive so the values aren't displayed in clear text in your logs. The most secure way is to directly integrate a secrets management service as a data source or a resource in Terraform. When the configuration is being deployed, Terraform can dynamically retrieve the sensitive data and use it in the configuration without it ever being stored even in environment variables. In each case, sensitive data in variables should be marked as sensitive and state data should be written to a secure location. State data will contain sensitive information stored in clear text. Be sure to properly store, secure, and encrypt that data as needed. Now let's head back to the configuration and make use of workspaces

Deploying the Development Workspace

Okay, back in the configuration. Let's go ahead and open up the commands m9, and before we do anything else, let's run terraform format. I'll go ahead and open up the terminal and run terraform format. Okay, now all of our files are properly formatted. Next, we'll run terraform validate to make sure our configuration is valid. Excellent. Our configuration is

valid. Now if you haven't already exported your environment variables for your AWS access key and secret key, go ahead and do that now. Scrolling down, we are going to create a new workspace called development. And why don't I expand the terminal a bit so we have some more room. The command is going to be terraform workspace new Development. Okay, Terraform not only created the new workspace development, it also automatically switched us over to the Development workspace context. If we look over to the left in our globo_web_app directory, there is a new directory called terraform.tfstate.d. If we expand that directory, we can see there is a Development folder in there, which will hold our state once we've run a Terraform plan and apply. If we'd like to get a list of all the existing workspaces, we can run terraform workspace list, and here it shows the default workspace and our Development workspace, and the asterisk shows which workspace is currently selected. With our Development workspace selected, let's go ahead and run terraform plan and send the output to m9dev.tfplan. I'll go ahead and copy this command and paste it down below. This is going to be a brand-new deployment, so it's going to have to create all of the resources we've defined in our configuration. It's a total of 24 resources that it is going to create. Let's kick off the terraform apply, and obviously this is going to take awhile, especially creating the load balancer, so go ahead and pause the recording now and resume once the environment has been created. All right, our deployment is successful, our development environment is up. Before we check out that public DNS link, I do want to point out that this is separate from your default deployment. If you still have that up and running, you're probably going to want to switch to the default workspace and destroy it so you're not paying for those instances. You'll also need to add a default key to the values in tfvars for the values that match what was deployed in the default environment. With that in mind, I'll grab the public DNS for our development environment and go over to a browser, paste that into the address bar, and after a few moments, there we go, our website is up in our development environment and you'll note that in the address it says development because it used that as part of the naming. Now let's try to deploy an instance of our UAT environment.

Deploying the UAT Workspace

We've created the Development workspace and deployed our development environment. Now it's time to create our UAT workspace and deploy it to the UAT environment. First we have to create the Terraform workspace, so we'll run terraform workspace new, and the name of the workspace will be UAT. Terraform has created that new workspace and switched us over to that workspace. If we expand the terraform.tfstate.d directory, we can see there's a development directory with a terraform.tfstate file in it and a UAT directory that is currently empty. Let's run a plan in our new UAT environment. We'll run terraform plan and send the output to m9uat.tfplan. Once again, this is a wholly new environment, so it's going to have to create all of the resources for that environment. Now the number of resources is different. It's 28 now, and that's because we're using the UAT environment. Remember, if we look at the tfvars file, for our UAT environment, we are deploying two subnets, so that stays the same, we're deploying slightly bigger instance types, t2.small, but now we're deploying four of those instances to be distributed evenly across those two subnets. Okay, with that in mind, let's go ahead and run the apply. Just like the previous deployment, this is standing up all new infrastructure, so this will take a little while. I'll pause the recording and we'll resume when all of the resources have been created. All right, our deployment is complete. Let's go ahead and grab that public DNS address, and we'll head over to a browser and paste that value in. And

after a few moments, there's the Globomantics test site for our UAT environment? If we take a look at the EC2 Management Console, there are now eight instances in the account. Two are from the default workspace, two are from the development workspace, and four are from the UAT workspace, and all of them are named in a way that is very obvious. If we select one of them and take a look at the tags associated with it, we can see that environment is set to UAT. So we could search on that tag within our account to find everything associated with the UAT environment. If you would like to set up and deploy the Production workspace, I invite you to do so now. The last thing I want to show you is how to select and destroy an environment so you can tear all of these environments down. We currently have the UAT workspace selected. Let's see how we select a different workspace. The command for that is terraform workspace select, and then the name of the workspace. In this case we'll select Development, and now we can destroy our development deployment by running terraform destroy -auto-approve. That will tear down the development environment, If you'd like to repeat that for the UAT environment, go ahead and do so, and you should definitely do that for the default workspace as well. You can also delete those workspaces with the delete command except for the default workspace, which cannot be deleted. Now we can tell Sally Sue we have successfully achieved her goal of multiple environments with a single configuration and different input values. Well done everybody!

Summary and Wrap Up

In this module we leveraged a single configuration to deploy and maintain multiple environments. This is like a new superpower, and it's one of the big benefits of using infrastructure as code. One thing we had to consider was the variances between the environments and make sure our input variables gave us flexibility to manage those variances with input values. We also looked into how we might deal with sensitive data like credentials and secrets. Finally, we got to use Terraform workspaces to manage our environments and state data using the special value terraform.workspace to control variable values. This is the final module in the course, and if I had to summarize things, some of the key points that I would like to bring forward are, number 1, it is so beneficial to build your infrastructure automagically. Removing yourself from the manual build process does a lot of great things for you. Most importantly, it ensures your deployments are going to be consistent and repeatable. Doing things manually leads to mistakes. We are fallible creatures, and that's just the way it is. When you codify your infrastructure, you're creating a way to consistently and repeatedly deploy that infrastructure. You're also creating configurations that are going to be reusable for different applications, especially when you use modules to take common patterns and abstract them to a shared resource that a bunch of people can take advantage of. Removing manual configuration and creating reusable configurations allows you to significantly boost your productivity. You're no longer bogged down in these manual processes. You're not troubleshooting these weird issues that were caused by someone making a mistake, and the reusable patterns means that you don't have to deploy the same thing over and over. You can do work that is more interesting and more challenging. Ultimately, it is about making your job better or, failing that, finding yourself a better job, and that's really what this course is all about. I want to help arm you with the skills to make deploying infrastructure less of a chore. And if all else fails, I want to help give you the skills to find the job that you want if you don't have it today. If you're looking for the next step in the world of Terraform, I would recommend checking out the other courses about

Terraform on Pluralsight. There are cloud-specific courses for Azure or AWS, or you can check out the deep-dive course. In fact, if you're planning to try for the Terraform Associate certification, this course and the deep dive should cover all of the content in the exam, plus a bit more. I'd also recommend checking out the other HashiCorp products like Vault or Packer. Vault is a secrets-management platform that can help solve the question of what to do with sensitive data. And Packer helps with creating gold images for use by something like Terraform. Lastly, if you'd like to get a regular stream of content from me, check out my YouTube channel, Ned in the Cloud, where I have an ongoing series called Terraform Tuesdays. That does it for this course. I want to take a moment to thank you for taking the time out of your hectic schedule to learn more about Terraform through this course. I hope you found the content valuable, and I welcome your feedback, suggestions, and comments. You can use the discussion boards found on Pluralsight, or you can ping me on Twitter, @ned1313, or on my website, nedinthecloud.com. I'm very easy to find. Until next time, go build something great.