Course Overview
Course Overview

Hi everyone. My name is Ned Bellavance, and welcome to my course, Implementing Terraform on Microsoft Azure. I am a Microsoft Azure MVP and founder of Ned in the Cloud LLC. HashiCorp's Terraform is a powerful tool to automate the deployment of your infrastructure across multiple clouds, both public and private. In this course, we are going to learn about the nuances of deploying infrastructure as code on Azure using Terraform and leveraging services in Azure to improve your Terraform automation. Some of the major topics that we will cover include using the Azure providers, including authentication and modules; using multiple providers in a single configuration; leveraging Azure Storage for remote state; and automating deployments with Azure DevOps. By the end of this course, you'll be ready to deploy and manage your infrastructure in Microsoft Azure with Terraform. Before beginning the course, you should be familiar with Microsoft Azure, especially the Infrastructure as a Service components and have an intermediate level of skill with Terraform. I hope you'll join me on this journey to learn more about using Terraform to deploy infrastructure as code on Microsoft Azure with the Implementing Terraform on Microsoft Azure, course at Pluralsight.

Using the AzureRM Provider

Introduction

Hey everyone, welcome to my course, Implementing Terraform on Microsoft Azure. There are few things that are more fundamental to Terraform than working with providers. Really, without providers, there's not a whole lot you can use Terraform for, so that is what we're going to get into in this module. Alright, what are we going to cover in this module? Well, the first thing that we're going to talk about is what are the prerequisites that you should be coming to this course with? It's very important that we sort of level set where you need to be in terms of your knowledge of both Terraform and Microsoft Azure. And then we're going to dive in a little bit into what the differences are between the way Terraform does things in their configurations and the way that Azure Resource Manager does things in deployment templates. Then we're going to introduce the Globomantics scenario that we'll be working with over the duration of this course, and finally, dive into some of the minutia of the different Azure providers. But first, let's talk a little bit about the course prerequisites. This is not a getting started course. I'm going to make some assumptions that you already have a base level of knowledge about how Terraform works in terms of both syntax and the concepts that exist within Terraform. So if you do not have a grasp on the basics of Terraform, you may want to do a getting started course first. In the same vein, I am assuming that you have a basic working knowledge of Microsoft Azure, especially the IaaS components of Azure, so things like storage, networking, and compute, as well as Azure AD. You should have a basic working knowledge of all of those things. If you don't have that core knowledge, then things might get confusing very quickly, so I would recommend doing a fundamentals course on Azure if you don't already have that knowledge. Now with that out of the way, let's talk a little bit about how Terraform does things versus the Azure Resource Manager template deployment.

Terraform for the Azure Admin

I'm going to go with the assumption that you are an Azure administrator, and over the tenure of your administration of Azure, you have come into contact with Azure Resource Manager templates. You've had to use them at some level, so I wanted to compare and contrast how ARM templates do things versus Terraform, and I think that will ease the transition to Terraform if you're used to using ARM templates. So at a very base level, ARM templates use JSON as their language, and Terraform uses HashiCorp configuration language. One of the nice things about HashiCorp configuration language is it supports comments, which the JSON and ARM templates don't really. Yes, you can add things as metadata, but just generally speaking, you can't just throw a hashmark and a comment in there. HCL is also a little bit easier for a human to read and a little more forgiving about syntax. A point of confusion when you're moving from templates to Terraform is what ARM templates call parameters, or the values that you submit to the template, Terraform calls variables, and the reason that could be a little confusing is because ARM templates do have a thing called variables, but those are values that are defined within the template itself, and Terraform's version of that is just called local variables that are defined in the locals configuration block. So a slight difference there in terms of verbiage, and it might trip you up a little bit when you're trying to make that transition. Fortunately, resources are just resources in the context of both the template and Terraform as are functions. The way that you invoke functions is a little bit different, but both have functions to deal with things like strings, numbers, and dates, and times. Another key difference between ARM templates and Terraform, in ARM templates, if you want to reference another template, say you have a really good template for rolling out a virtual network, you would just reference it using a nested template reference. In the world of Terraform, there is a similar concept except it's called modules. So in the base configuration for a Terraform deployment, you would reference a VNet module, let's say, and supply that VNet module with the proper variables to create that virtual network. So the relationship is very similar, the way that you invoke it is a little bit different. Another important difference between the two is ARM templates rely on explicit dependency. If you're creating a virtual machine in Azure and that virtual machine has a virtual NIC that you're also creating, you have to explicitly say that that virtual machine is dependent on the creation of that virtual NIC. Terraform will build a dependency graph automatically and determine, oh, if I'm going to create that virtual machine, I have to create that NIC first. That's really nice that you don't have to always put in these explicit depends on statements. And then finally, when you're referring to those resources in an ARM template, you usually do that through the reference function, or the resource ID function, or sometimes, you just concatenate the name of the resource. In Terraform, it's a little bit simpler. You simply refer to the resource by using resource addressing, or you can pull information from the Azure provider using a data source and refer to that data source. So I think it's a little more simpler and clearer in Terraform when you want to refer to an existing resource or a data source. Speaking of moving from ARM templates to Terraform, let's take a look at our scenario.

Welcome to Globomantics

For our scenario in this course, we are going to be helping out at Globomantics. And what is Globomantics? Well, they are a logistics company for restaurants across the globe, so they help get that taco from the supplier to the restaurant and into your eager hand, and the way

that they do that is through applications. And those applications are hosted in an on-premises data center today. Now Globomantics has just signed a couple really big contracts with new restaurant providers, and they're realizing that they don't have the capacity they need in their existing data center. So they have chosen to migrate some applications up to Microsoft Azure, and you are going to help them make that goal a reality. Now who is on the team for this project? Well, we've got a few different players going on. On the cloud side, we've got Chris Jones. She is a cloud architect, and she is going to be helping build the infrastructure that the applications are going to be running on. We've also got Danny Brown, and he's a security administrator. He's got some concerns about how things are going to be set up in Microsoft Azure. He wants things to be deployed consistently and repeatedly, which makes Terraform a pretty good fit, and he also wants to insert his own security tools in the process. And then finally, we've got Hector Sanchez who is a software engineer, and he is going to be deploying applications in the environment that Chris Jones creates. So those are the players at Globomantics that we are going to be working with to get this project complete. This group has collectively decided to use Terraform, and in order to interact with Azure, they are going to need to use the Azure providers within Terraform.

Azure Providers

There are actually three different providers for Azure in Terraform. One is the basic Azure provider, and this is the one that you will use to interact with Azure public cloud, Azure gov cloud, or one of the sovereign clouds. It uses the Azure Resource Manager API. There used to be an older Azure provider that used the service management, but ASM and the provider have both been deprecated. Another provider that is very similar to the Azure provider is the Azure Stack Provider. If you're not familiar with Azure Stack, it's an on-premises extension of Microsoft Azure, and it uses the same ARM API, but some of the versions and resources are a little bit different, so it actually did require its own provider. And lastly, there is an Azure provider to deal explicitly with Azure Active Directory. That is a relatively new provider. There used to be Active Directory components in the regular Azure provider, but Microsoft realized with the complexity of Azure Active Directory, it probably needed its own provider, so now it has one. Now just to refresh your memory a little bit, what goes into a Terraform provider, well, first of all, Terraform providers are versioned, as newer versions of the providers are released, you can specify what version you want to use, and there's actually a big 2.0 release of the Azure Provider that you might not want to jump up to right away, so it's good that versioning is an option. Providers have data sources, and this is information that you can pull from the provider about your target environment. So if you're using the Azure Provider, you might want to pull a list of marketplace images or get an existing virtual network that's already been provisioned. The other half of that equation is resources, and resources are things that you can create in the target environment, so for instance, you might want to create an Azure virtual machine in a VNet that you got from the data sources. While it's not specifically part of the provider, most providers have modules associated with them that help you easily deploy common configurations for that provider, and those modules can be found on the public Terraform Registry, which is registry.terraform .io. and then finally, most Terraform providers have some form of authentication you use to interact with that provider. In the case of the Azure providers, you're using Azure Active Directory to provide that authentication. Now there are a number of different ways to perform that authentication against Azure Active Directory, so let's take a look at those now. When you're authenticating

to the Azure provider or the Azure Active Directory provider, there are a number of different ways to go about that. You can use the Azure CLI. If you do az login from the Azure CLI, Terraform will be able to find those credentials and use them when you're invoking the provider. Another way to go about authentication is by using a managed service identity. Managed service identities can be assigned to objects that run within Azure, so say you're running an Azure virtual machine that has Terraform installed. You can tell Terraform to use the managed service identity of that virtual machine for deployments as opposed to using the Azure CLI for authentication. Another potential option is using a service principal with a client secret, so you can sort of think of this as a username and password. You create a service principal in Azure Active Directory, and you generate a client secret for it, and then you can use those two items to authenticate to Azure Active Directory. Now you might not be entirely happy using a client secret. You might want to use a certificate instead, so the fourth option for authentication is to use that same service principal, but use a generated client certificate for authentication instead of using a secret or password. So let's take a look at an example of how you would use this with the Azure provider.

AzureRM Provider Examined

Now you note at the top of the screen, it says AzureRM Provider. And as I mentioned before, there used to be an Azure Service Manager Provider, which was deprecated along with the Service Manager model in Azure. Now everything uses Resource Manager, but the actual name of the provider when you're invoking it is azurerm. So the way that you would use the azurerm provider is to create a configuration block, which is a fundamental component of Terraform. In a configuration block, you first have to say what type of object you are creating that block for, so the provider keyword lets Terraform know we're provoking a provider, and then the first to label AzureRM tells it which provider we're using. Within the configuration block, you can specify a version argument saying what version you want to use. And in this case, that little squiggle with the greater than sign says I want to use version 1.X, so I want to stay in the 1 major version, but I will do 1.2, or 1.3, etc. Because version 2 is coming out soon and there might be breaking changes, this might be a good thing to put in configurations so you stay on the 1.X provider until you're sure you're ready to make the jump to 2. You can also specify an alias in your provider, and the alias allows you to create multiple instances of the same provider, and that's something we're going to deal with in the next module. You could specify the subscription ID that corresponds to what subscription you're going to be using for that provider, and then you can specify some authentication information. If you wanted to use the service principal and the client secret, you would specify a client_ID, which is the ID of that service principal, and then client_secret, which is the password for that service principal, so this is an example of using that configuration block with pretty much everything specified out. You can also specify some of these values using environment variables. If you wanted to use environment variables to specify the values, there are many Azure resource provider environment variables, so let me just go over a few briefly. You can specify the client_id and client_secret by using ARM_CLIENT_ID and ARM_CLIENT_SECRET. Please note that these are case sensitive. You can use these two to specify that principal ID and secret in the environment variable instead of including it in the configuration. You can also specify what environment you're using whether you're using Azure public cloud, gov, or one of the sovereign clouds. You can specify the SUBSCRIPTION_ID, the TENANT_ID, which is the Azure Active Directory tenant that has the service principal you're

using for authentication, or you could specify an environment variable ARM_USE_MSI, and that tells Terraform to use the managed service identity on the system where terraform is running. So these are just some of the different environment variables, but I did want to bring them up because we're going to be using them in some of our configurations instead of putting the information in that provider block. So what are we going to be using this provider information to deploy? Good question.

Deploying a Virtual Network

In our Globomantics scenario, one of the first things that they want to get going is a virtual network to do some application testing. So you, as Chris Jones, are going to be setting up an environment in Microsoft Azure, and basically, you're just going to be creating a virtual network in Azure, and you're going to create two subnets in that virtual network, one called web and the other one called database. Now you don't have Terraform or the Azure CLI already installed in your workstation, and you'd like to get this going pretty quickly, so you've decided to use Azure Cloud Shell to perform this deployment, and the reason is Azure Cloud Shell already has Terraform installed, it already has the Azure CLI installed, and in fact, it takes care of the authentication piece for you because it uses the Azure Active Directory credentials you used to log into the portal with. That makes the deployment process much simpler. So let's go ahead and deploy that virtual network. For our demo, just a few quick things before we get started, we are going to take a look at the Terraform file, review what's in there, we're going to deploy that configuration using the Azure Cloud Shell, and then review the results and make sure that our virtual network creates successfully. If you want to play along, you really only need two things. You're going to need an Azure subscription, obviously, and you're also going to need the demo files, which you can find on my GitHub account. It's Ned1313 and then just look for the course name, or you can find it on the Exercise Files tab on the course web page. One final note of caution, some of the resources we're going to deploy in this course may cost money. You're deploying resources in Azure, and while I've tried to keep that cost as low as possible, there are going to be resources that may cost some small amount of money, so I just want to say that up front. I'm trying to keep the cost low, but just understand that your Azure subscription may incur some cost to you, so consider yourself suitably warned. Alright, let's go over to our demonstration environment and get started.

Reviewing the Virtual Network Config

Alright, here we are in Visual Studio Code, and we are going to review the configuration for rolling out this virtual network. You'll note I have the exercise files in the left pane, and in the first directory, 1-main-vnet, I have the main.tf file open. So now that we see that, let me just shrink that up to give us a little more room to work. Alright, first, we're going to define some variables. This should be very familiar if you've ever used Terraform before, but we're going to set up a resource_group_name, and I've set type equal to string, but I haven't given it a default value, so you're going to have to specify that when you run Terraform plan. For the location, I have set a default of eastus, but you can obviously change that to whatever location makes sense for you. And then scrolling down, we are setting the IP address range for the virtual network, and in this case, it's set to a default of 10.0 .0 .0 /16, and directly below

that, we are setting the subnet_prefixes that are going to be in that virtual network. And you'll note that the type is a list of string, so you do have to specify a list here. I've set the default to 10.0 .0 .0 /24 and 10.0 .1 .0 /24. Please note that these are inside the VNet side of range, so you want to make sure that you do that if you don't go with the defaults. And finally, we are setting the subnet_names. That is also a list of strings, and we've got web and database in there. If you're going to change any of this, just make sure you have the same number of subnet_prefixes that you have of subnet_names otherwise you're going to get an error. Now let's scroll down to our provider definition, and that's it. We say provider azurerm, and the rest is going to be magically filled in when we're using the Azure CLI, and I'll show you how that works in a moment. Now let's get down to our resources. For our resources, we are using a module to deploy our virtual network, and we're specifying the source as Azure/vnet/azurerm, so where is that source? That source is on the Terraform public registry. Let's go over to a browser and take a look at that. If you go to registry.terraform .io, it will take you to this Terraform Registry. And if we scroll down a little bit here, it shows us the different providers that you can look at modules for, and boom! There's Azure! So here are the modules associated with the Azure Provider. Let's scroll up a little bit, and we can tick this little box here that says Verified, so it will only show us modules that have been verified by Microsoft as working for the Azure Provider. And the one that we are actually using is the vnet, so if we go to the next page, there is the vnet module. We can click on the vnet module, and it gives instructions for how to use this module, what the expected inputs are, and what the expected outputs are, and what resources actually get provisioned. So this is the module that we are invoking in our configuration. Let's go back to the configuration and continue. So within the module configuration, we're specifying a resource_group_name; the location; the name of our vnet; which we're setting to be the same as the resource_group_name; we're setting an address_space, the subnet_prefixes, the subnet_names; and then there's an nsg_ids property that needs to be set to an empty map otherwise it throws an error. If you did have network security groups that you wanted to associate with the virtual network, you could provide a map of them here. We don't at the moment, so we're going to leave that empty. And then lastly, we're specify two tags for this virtual network. We want to set an environment tag equal to dev and the costcenter equal to it. So that's all of our resources. If we scroll down a little bit more, we get to OUTPUTS, and what we are getting as an output is the virtual network id, which is an output that's exposed by the module, so we're referencing it by doing module.vnet - main.vnet_id. So that's the whole configuration. There's not a whole lot going on here. So let's go ahead and launch the Cloud Shell and actually deploy this configuration.

Deploying the Config with Cloud Shell

Okay, here we are in the Microsoft Azure Portal, and you'll note that we're logged in as Chris Jones at Globomantics, and we'd now like to launch the Azure Cloud Shell to deploy our configuration. So all we have to do is click on the Cloud Shell, and it will launch a Cloud Shell below. Now if you've never used Cloud Shell before, it's actually going to prompt you first to create a storage account where it's going to store any files that you create within the context of the Cloud Shell. Now Chris has logged in the Cloud Shell before, so she is not getting prompted for that. It will also prompt you whether you want a Bash or a PowerShell, and you can change it by clicking on the drop-down and selecting PowerShell instead of Bash. But for our purposes, we are going to be using the Bash version of PowerShell. Now let's go ahead

and click on the Maximize button so it takes up the whole screen, and what we're going to do first is create a folder to store our Terraform configurations. So we're going to run mkdir terraform to create that directory, we'll go into that directory, and we're going to create a second directory called 1-main-vnet and go into that directory. Alright, now we're all set with our directory structure. Now one of the things that I mentioned is that Terraform is going to be using the credentials for the Azure CLI, but we didn't log into the Azure CLI. The Cloud Shell does that automatically for us. So if we run az account show, it is going to show us which subscription we are using, which is the PS subscription, and it's going to tell us what user we're already logged in as, so we're logged in as Chris Jones. So as part of the Cloud Shell spin up, it logs you into the Azure CLI automatically. The other things is if we run terraform version, you can see that Terraform is already installed with version 12.10, which is the current version of Terraform. Sometimes the Cloud Shell is one minor revision behind. They do try to keep it as up to date as possible. Right now, it's current. If it's not, Terraform will say, hey, there is a newer version, and this is the newer version, but in Cloud Shell, you're not able to install applications. It's all preconfigured for you. so we validated that we're logged in through the Azure CLI. We validated that Terraform is already installed. Now let's go ahead and create our configuration in Cloud Shell. Fortunately, there is a code editor in Cloud Shell, so we can run code to invoke that code editor, and give it a file name that we want to work with, so we'll call it main.tf, and it launches the code editor above the Cloud Shell. That's pretty convenient. And this is based off of the same code that Visual Studio Code uses. Let's go back to Visual Studio Code, and we're going to select the contents of our main.tf configuration and copy it, and then we will paste it in here and do Ctrl+S to save the file, and you can see it actually adds syntax highlighting throughout the document, which is pretty convenient. Now that we have main.tf file saved, we can do Ctrl+Q to close out of the editor, and I'm just going to do clear to bring the command all the way back up to the top of the screen so it's a little easier to see. The next thing to do is to go through the Terraform core workflow, which starts with terraform init. Terraform init will pull down the module that we're referencing within our configuration, as well as add the provider plugin for the AzureRM Provider, and you can see, it pulled down version 1.2 of the Azure vnet module, and it also pulled down version 1.35 of the azurerm provider. So we've successfully initialized our configuration. Let me just clear it out again so we can get back to the top of the screen. The next thing to do in our configuration would be to run terraform plan, and then we do need to specify a value for the resource_group_name, so we're going to set that to vnet-main, and then we're going to save the plan to a file vnet.tfplan, so let's go ahead and run that now. And there we go! It runs pretty quickly. We can see it's going to add four new resources. If we scroll up a little bit, we can see it's creating a resource_group, it's creating two subnets, web and database, and if we scroll down a little bit more, we can see it's creating a virtual network to house those two subnets. Scrolling all the way down, we can see there is one warning that was thrown here, and it's telling us that some property inside the module is going to be deprecated. Now we don't have to do anything about this. It's just a warning. And Microsoft is responsible for the upkeep of this module, so we're going to assume they're going to swap out the network_security_group for that azurerm_subnet_network_security_group_association resource at some point in the future before it's fully deprecated. Now that we've successfully run our plan, we can run terraform apply vnet.tfplan. Go ahead and copy that, and we'll paste it here. And now it's going to go ahead and create that resource_group, the virtual network, and the subnets within that virtual network. Fortunately, creating a virtual network doesn't take very long, so this should go relatively quickly. Alright, there we go. It has successfully created our resources, and you can see that the vnet_id output is there at the bottom of the

screen. Now if we want to go ahead and minimize this Cloud Shell and refresh our view of the resource groups, we can go ahead and confirm that the vnet-main resource group is there, and we'll click through on that. And within that resource group, there is a single resource, the vnet-main virtual network. We can click on that, and we can see that the tags were successfully applied. We've got costcenter : it, environment : dev. It's using the correct address space that we specified. And if we go to Subnets, we can see that the database and web subnets have been created with the correct IP address range. So we have successfully deployed our first Terraform configuration in Microsoft Azure using the Cloud Shell. Good stuff.

Summary

Alright, what did we learn in this module? Well, we looked at the differences between the way Terraform does things versus an ARM template. And I hope by looking at the configuration, you got a feeling for how much easier it is to read through a Terraform config than parse an ARM template. That's not a dig on ARM templates. It's just the nature of JSON. It's a little bit harder to read. Terraform is just a little more human readable. We also talked about the three different Azure providers that are out there and dug into how to use the AzureRM provider specifically, and then we talked about the different ways that you can authenticate against the Azure providers and went through an actual deployment of resources using the AzureRM Provider. Now that's just the beginning of our journey. Coming up, the security team and Danny Brown are going to get involved, and that means we're going to have to deal with multiple subscriptions in Microsoft Azure and how to use service principals for authentication when using the providers. That's all coming up in the next module. I'll see you there.

Creating Multiple Providers

Introduction

As your configurations grow more mature and complex, there is a really good chance you're going to need to use multiple providers within the same configuration, so that is what we're going to be talking about in this module. Hey everyone, this is Ned Bellavance. I'm a Microsoft Azure MVP, and this is Creating Multiple Providers. What are we going to talk about in this module? Well, the first thing that we're going to talk about is the multiple providers in Azure and how you could instantiate them, and what I mean when I'm saying multiple providers because there could be two different ways to take that. And then we're going to dig in a little bit about how to use the Azure AD Provider. In the previous module, we saw how to use the AzureRM Provider, and now it's time to learn a little bit about the Azure AD Provider and how it might be slightly different. Then we are going to dive into our Globomantics scenario and get the security team connected to the environment that we have started to configure. But first, let's talk about multiple providers in Azure.

Using Multiple Instances

As I mentioned in the previous module, there are three different Azure providers, there's Azure, Azure Stack, and Azure Active Directory. Now we aren't going to be using the Azure Stack Provider in this course because, well, it's a lot of work to set up an Azure Stack development kit, but we are going to be using the Azure and Azure Active Directory Providers, and we'll see why in a moment. Now when I say, multiple providers, that can have more than one meaning. I could simply mean that we're using both the Azure and Azure Active Directory Providers. That would be multiple providers. You might have a configuration where you want to create, say, an Azure function, and then you also want to create an Azure service principle and role for that function to assume. I could also be talking about using multiple instance of the same provider, same more than one Azure Provider. There are two primary reasons to use multiple Azure instances. The first is you're going to work with more than one subscription in a configuration. Each instance of the Azure Provider is specific to a subscription. The second reason would be if you wanted to use multiple authentication sources within your configuration. Say you wanted to use the managed service identity for some resources and a service principal for others, that could be a reason to have multiple instances of the Azure Provider. Now let's take a look at what's going on with the security team and see which of these two they need for their configuration.

Security Network Deployment

Danny Brown is a security administrator on the info sec team, and he is responsible for rolling out the security toolsets in Microsoft Azure, so he has set up a separate security subscription to place all of the security related resources and lock down access to that subscription. Within that subscription, he wants to deploy a virtual network to house some of the security tools that they are going to use for a data analysis and vulnerability assessment. Now the resources that will generate this information are going to be living in other virtual networks, so what Danny would like to do is have all of those virtual networks have a peering connection to that security-vnet so that it can collect information for analysis. So what Danny is asking is anybody who spins up a virtual network within Azure, he wants them to be able to create that peering relationship to the security-vnet. Fortunately, all of these subscriptions within the Globomantics environment are using the same Azure Active Directory tenant. And because of that, Danny is going to create a service principal within that Azure Active Directory tenant, and he's going to create a custom role on the security subscription that allows that service principal to create VNet peerings, and he's going to give the credentials for that service principal to the team that is responsible for spinning up these virtual networks, so they'll be able to do the peering on their own without Danny having to get involved. In order to deploy this configuration, Danny is going to have to use both the AzureRM Provider and the Azure Active Directory Provider, so let's take a look at an example of using the Azure Active Directory Provider.

Using the Azure AD Provider

Just like the AzureRM Provider, the Azure Active Directory Provider starts with a provider block, and then you specify the provider type as azure_ad, and then you can also specify a version. Now you remember, I mentioned that the azure_ad provider is somewhat new to the

scene, and for that reason, it's not even at a 1.0 release. The current version is .6. Now you could skip the version here and just take the latest if you wanted to. You can also specify a subscription_id, and more importantly, you can specify the tenant_id. Because we're working with Azure Active Directory, we're going to be working with a specific Azure Active Directory tenant. And then within the configuration block, you could also specify the client_id of a service principal to work with this provider. Now you might not want to specify the client secret within the provider block. Maybe you want to use environment variables. Well, the good news is, just like the azurerm Provider, the azure_ad Provider also uses environment variables. In fact, it kind of uses the exact same environment variables that we looked at before, so things like ARM_CLIENT_ID, ARM_CLIENT_SECRET, the ARM_ENVIRONMENT, SUBSCRIPTION_ID, TENANT_ID, And ARM_USE_MSI. All of those same environment and variables also apply to the Azure Active Directory Provider. So if you specify them for one, you're really kind of specifying them for both. If you want to override one of these environment variables, you can specify it in the provider block, and that actually takes precedence over anything that's in the environment variable. So let's go over to our demonstration environment and help Danny step through the process of spinning up his security-vnet and that service principal.

Reviewing the Security Configuration

Okay, here we are in Visual Studio Code, and you'll see I've got the directory 2-sec-vnet open and the main.tf file open in that directory. Let's go ahead and minimize that down and give ourselves a little more room. Now we've got the standard variables in our configuration, we're defining things like the resource_group_name, the location, and then we get down into the virtual network settings. For the vnet_cidr_range, I've set a default of 10.1 .0 .0 /16. What I want to point out is the point of this exercise is to create a peering relationship between this security virtual network and the main virtual network that Chris Jones has set up, so you want to make sure that whatever you choose for this IP address range does not overlap with the other virtual network that's already been created otherwise the peering relationship is, well, it's not going to work very well. Then we're setting our subnet prefixes for the security subnet. I'm using 10.1 .0 .0 /24 and 10.1 .1 .0 /24 and then giving a name to each of those subnets, siem and inspect. So those are the variables for our security deployment. Let's scroll down a little bit more, and we have a DATA section here, and we are pulling information from Azure about our azurerm_subscription. And what this data source does is it pulls the Azure subscription information about the subscription that's being used along with the azurerm Provider so that we can get information like the ID of the subscription, and we're going to be using that information a little bit further down. In the PROVIDER section, if we scroll down a bit, we are establishing two providers here, the azurerm and azuread providers. Now because we are using the Azure CLI for authentication, all of the information that we need for each provider will be provided through the Azure CLI authentication, so we don't actually need anything in the RESOURCES block of these providers. Let's scroll down a little bit more into our resources just to review quickly what we're trying to create here. We're trying to create an Azure virtual network, and we're also trying to create a service principal in Azure AD and assign it a role on the security subscription. So this first section just deals with the networking aspect of it. The first thing we're doing is creating an Azure resource_group, and we're giving it a name based off of the resource_group_name and a location based off the variable location, and we're tagging the resource group security for the

environment. Scrolling down a little bit more, just like the configuration from the previous module, we are using the vnet module from the Terraform Registry to create our virtual network. The only thing that's really different about this configuration is the tags, so we are setting the environment equal to security and the costcenter equal to security, so we've basically already seen this portion of the configuration before when we were spinning up the virtual network for Chris Jones. Now scrolling down a little bit further, we get into the creation of our azuread_service_principal, and we need to create a number of resources to get this done. The service principal will need a password, so we're using a special resource called random_password, and random_password does exactly what you think it does. It creates a random password, and within the RESOURCE block, we tell it what the length should be for that random password, we're making it 16 characters long, and whether or not it should use special characters. We're setting that to true. If you've worked with service principals in Azure Active Directory before, you'll know that you actually have to create an Azure AD application for that service principal, and so the next resource is an azuread_application, and we're giving it the name vnet-peer. Then we can create our azuread_service_principal, and the only thing we need to specify in there is the application_id. We retrieve that by using resource addressing to refer to the attribute application_id. So the way that we do that is we refer to the resource azuread_application, the name label on that resource, vnet_peering, and then the attribute that we want from that resource, application_id. So if you're used to ARM templates, this is a different way of getting to an attribute than you would use in an ARM template, and I thin it's a little bit more straightforward as well. Scrolling down a little bit more, now that we have our service principal, we can assign it the password that we generated from random_password. So within this resource, we are specifying the principal_id of this azuread_service_principal, we're specifying the value from the random_password resource, and the attribute for that is called result, and then we have to set a date for when this password will expire. For that value, we're using 17, 520 hours, which is roughly 2 years. Now that we have a service_principal, we can create a role in the security subscription and assign that role to this service_principal. So first, we have to define the role. So let's scroll down a little bit. We are creating an azurerm_role_definition, we're naming the role allow-vnet-peering, and the scope defines where is this role available. For the scope, we are using the data source azurerm_subscription. So the way the resource addressing works for that is we have to say this is a data source, so start with data, dot the data source type, azurerm_subscription, the name label on that data source, .current, and then the attribute we want from that data source, in this case, .id. Then we have to give a list of permissions for this role definition, and those permissions are made up of actions and not_actions. If you search Microsoft's documentation, they outline exactly what actions are necessary to establish a network peering and modify it, so I simply took that list of actions and pasted them in here. Further down, we get to assignable_scopes, which is where can this role be assigned, and once again, we're referring back to the subscription_id of the current subscription we're working with, which should be the security subscription. Now we have our role definition created. The last step is to assign the role to the service_principal that we created and set it to the scope of the virtual network that we created. So the three arguments for this role_assignment resource are, first, the scope, where am I assigning this role, and we're setting that to module.vnet - sec.vnet_id. So that's the resource id of the virtual network that we created. Then we have to specify the role definition, and so we're grabbing that from the role_definition resource, and finally, the principal_id, which is the ID of the service principal that we want to assign this role to. Now the whole point of this was to take this service principal and give it to Chris Jones and anybody else who is going to be creating

virtual network peerings to the security_vnet. So if we scroll down a little bit more, we get to a PROVISIONERS section. There is a special resource in Terraform called the null_resource. And basically, the whole point of the null_resource is that you want to run a provisioner, but it's not necessarily triggered by anything specific. You're not spinning up a virtual machine or something like that, so the null_resource allows you to embed a provisioner in your configuration. We are using the local-exec provisioner, which runs a command locally on the system that's invoking the Terraform script. And basically, all we want this command to do is write all the necessary values that Chris Jones will need for her step out to a text file. So we're running a number of echo commands and sending the output to a file, next- step.txt. We are sending the vnet_id, the vnet_name, the subscription_id for the security subscription, the application_id for the service_principal, the principal_id for the service_principal, the password for the service_principal, and then finally, the security resource group name. We will need all of that information to set up the vnet_peering in the next step of the process. Finally, we are also sending some of this information to OUTPUTS, so the vnet_id, the vnet_name, the service_principal_client_id, the client_secret, and the resource_group_name are all being made available as outputs as well. So that's the entire configuration. Now functioning as Danny, let's go ahead and deploy this configuration to the Azure environment.

Deploying the Security Configuration

Alright, we are ready to deploy our configuration to the Azure environment, but where are we going to deploy this configuration from. Fortunately, Danny Brown has taken the liberty of using the Terraform marketplace item to spin up a virtual machine in the security subscription that has Terraform preinstalled and also creates a storage account for the storage of remote state for Terraform configurations. If you're looking for that marketplace item, we can go over to the browser, and here we are logged in as Danny Brown. If we go to Create a new resource and go ahead and type in Terraform, there is a Terraform item right here, and this takes you to the marketplace item that you can click through Create, and basically, what it does is create a virtual machine in your environment that has Terraform preinstalled, the Azure CLI preinstalled, it actually creates a managed service identity for the virtual machine if you want to use that, and it also installs some helper applications like Unzip, JQ, and apt-transport-https. Now I've already gone through the process of deploying this virtual machine. And if we go back to Visual Studio Code, I have an SSH connection to that virtual machine. So I'll go ahead and pop open the terminal down below, and this is our SSH session into that virtual machine. If we go ahead and run terraform version, you can see we're running 12.10, so this is the same version that we were using with Chris Jones in the Azure Cloud Shell. If we do an ls of the current directory, you'll see there is a file and directory there, the tfEnv.sh, which allows you to assign that MSI for the virtual machine contributor rights in the subscription, We don't need to do that. There is also a tfTemplate directory. Let's go into that and look at what's in there. There is an azureProviderAndCreds.tf and a remoteState.tf, and basically, the idea is you could take either of these files and add it to your existing configuration to include the Azure Provider or include the remoteState configuration. Let's take a look at what's in that remoteState.tf file. Basically, what it is is a backend configuration that uses AzureRM as the backend. It includes the storage account that was created by the template, the container_name, which is also created by the template, and the key to store in that container name, and the access_key for that storage account. So all you have to do is include this remoteState.tf file in your configuration, and it will be set up to use this storage account as

the remoteState backend, and we're going to do exactly that in a moment. First, let's go up one directory, and we'll create a directory to house our Terraform configuration. Alright, we've created that directory. We'll navigate into it, and now we're going to copy that remoteState file to our local directory. There we go. We now have that remoteState configuration. We also want to add our main configuration, so we will use vi and create a file called main.tf, and all we're going to do is copy the entire main.tf configuration, and we'll hit I for insert and paste in the configuration here. There we go. Hit Escape and then :wq to write the information to file and quite vi. There we go. If we look at the directory, we now have our main.tf and our remoteState.tf. Now that we have our configuration ready, the next thing to do is log into the Azure CLI so that we're using Danny's credentials for the deployment of this configuration. So all we have to do is az login, and now it's going to give us a web page to go to, so I'll go ahead and highlight that, copy it, go to the browser, paste that address in, and now we'll ask for that code, go ahead and grab that code, paste it in here, and it's going to ask me to pick an account to sign in with. I'm already signed in with Danny Brown's account in the browser, so I can simply click that. That completes the sign-in process. And if we go back to Visual Studio Code, it has now logged us in, and we can see that we are using the security subscription names Sec, and we are logged in as the user DBrown@ globomantics.xyz. So when we invoke the configuration, it's going to take the information that's cached by the Azure CLI and use that for both the AzureRM and AzureAD providers. So let's go ahead and start with terraform init. And let me just maximize this so you can see what it just did. It initialized the vnet module by downloading it from the Terraform Registry, it initialized the storage backend, so it was able to successfully connect to that storage account backend, and it's going to use that for storage of the state file, and it downloaded the providers for null and random, which means it already had the AzureRM plugin downloaded from when I was working on this earlier. Now the next step is to run terraform plan, but we'll run terraform plan sec.tfplan, and so this should tell us what resources it's going to generate. Okay, it has successfully run our plan, and we can see it's adding 12 new resources, so that is the virtual network along with all of the Azure AD components and that role that we wanted to create. So if we want to go ahead and create all of those resources, all we have to do is run terraform apply sec.tfplan. We'll go ahead and run that now, and it's going to go ahead and create those resources. Now while those are creating, we can jump over to the portal and verify that they're creating successfully. Alright, so we're back in here. Let's go into our Resource groups, and we can see that there is now a security resource group. If we go into the security resource group, there should be a virtual network. There we go. There is the security virtual network we are looking for. We can go ahead and click on that. And if we look at our subnets, we can validate that the inspect and siem subnets have been created. We can also go to Access control. And if we look under Access control in Role assignments, we can see down at the bottom there is a custom role called ALLOW-VNT-PEERING, and the app vnet-peer has been assigned that role. So it looks like everything provisioned successfully. Let's go back to Visual Studio Code, and we can see the outputs are there. If we take a look at what's in the directory, there is our next- step.txt file. Let's take a look at what's in that file, and there we go. It exports a number of values as environment variables for the next steps in the VNet peering process, which will be executed by Chris Jones. So the last step here is for Danny to securely send this information over to Chris Jones so she can move forward with the VNet peering process. So let's take a look at what Chris Jones is going to have to do.

Multiple Instances for Network Peering

Chris Jones is going to be setting up a virtual network peering between the virtual network in the development subscription main-vnet and the security-vnet in the security subscription. So already, you can see we are working with two different subscriptions here, which means we're going to need at least two instances of the Azure Provider. In order to successfully create this VNet peering, she is being supplied with that service principal that has permissions to create VNet peering on the security-vnet side, but one side is not enough for a peering. The peering configuration has to be set up on both virtual networks for it to be successful. So Chris is going to create a role on the dev subscription that mirrors the role that was created on the security subscription and assign that role to the service principal, and basically, the service principal will now be able to create the virtual network peering on both subscriptions thereby creating a successful network peering connection between the two virtual networks. Now how are we going to go about setting up multiple azurerm providers? Let's take a look at some example provider configurations. There is the very minimal azurerm provider configuration where all data is being provided through environment variables or the Azure CLI. And if you don't specify a provider for a resource or a module to use, it's going to assume that this is the provider that you want to use in your configuration. If you do want to get more specific about a particular provider, within that provider, you can specify an alias, say, security, for instance. Now we can refer to this provider directly by specifying its alias. Also, within the provider, you can tell it what subscription_id to use this provider with. And if we wanted to set up another provider with a different subscription, we could invoke the same configuration block, use a different alias, say, peering in this case, and give it a different subscription_id, and that's exactly what we're going to do in our configuration. Now you might be wondering, how do you reference these alias providers in the configuration? I'm glad you asked. So if you are creating a resource, let's say you're creating a resource group, the way that you would reference the provider is by adding a provider argument to the resource block and then referencing the AzureRM Provider by putting in azurerm. whatever the alias is for that particular provider. So if we wanted to reference the security provider, we'd put azurerm.security. With modules, it's a little bit different because when we think about a module, that's a full-blown Terraform configuration on its own, and it may have multiple providers within it, so you need to be able to reference each of those providers with the alias provider you want to use. And the way you do that is by submitting a map object to the module using the providers argument instead of provider, so you'll notice that providers if plural because there could be more than one provider in the module, and each provider in the module is going to have some name associated with it. If it's just the vanilla azurerm provider, you can refer to it by saying azurerm as the key and then for the value, the aliased provider you want to use from your main configuration, in this case, azurerm.security. So those are the ways to refer to a provider alias when you're provisioning resources and modules. So now, let's dive into the configuration and get ready to create that virtual network peering with Chris Jones.

Reviewing the Peering Configuration

Okay, we're back in Visual Studio Code, and you'll see I've got the directory 3-vnet-peering open, and in that, there is a vnet- peering.tf configuration. This is the configuration that Chris Jones is going to use to create that VNet peering connection between the main-vnet she has

and the security-vnet that was created by Danny Brown. Now if we look in the middle pane, let me just hide the EXPLORER so we have a little more room, we have a number of variables defined that we're going to need to use to successfully create this peering connection, and you might notice that these variables are the output from the next- steps.txt file, so things like the sec_sub_id, sec_client_id, client_secret, the sec_vnet_name, vnet_id, the sec_resource_group, the sec_principal_id, all of those are part of that next- steps.txt document, and we'll be getting back to that in a moment. If we scroll down a little bit more, you can see we are fetching the current subscription that's selected from the default AzureRM Provider because we're going to need that subscription_id a little further down, and then we're defining our two providers. We've got a security provider and a peering provider, so let's look at the security provider first. So we have defined an alias of security. For the subscription_id, we're giving it the sec_sub_id, and then we are giving it the client_id and client_secret of that service principal that we created in the previous step. There are two more values in here, which are skip_provider_registration and skip_credientials_validation. The reason those are in there is that the service principal that we created has very restricted rights on the security subscription. It only really can do VNET peering, and that's it. So if it tries to enumerate all the providers for registration on the Microsoft Azure side, that is going to fail and throw an error. So by putting these two values in, we're just avoiding that error getting thrown. For the next provider, we are giving it an alias of peering, so this is the provider we're going to use to establish the peering on the main subscription, and then we're giving it the subscription_id of the main subscription, which will be the subscription that's being used by the main azurerm provider, and then we're giving it the client_id and client_secret of the service principal, and then finally, those two skip_provider and skip_credentials values. So those are our two additional providers. We still have our original AzureRM Provider that doesn't have any of these values in it, and that's going to use what's stored in the Azure CLI for its values, so we actually have three instances of this AzureRM Provider within the configuration. Scrolling down a little bit more, we get into the azurerm_role_definition, and basically, what we need to do is create the same role_definition for our main subscription that we created for the security subscription, so all the settings in here are basically the same except the subscription scope is the main subscription, and the assignable_scopes is the main subscription, and the name is a little bit different so we don't have a collision of roles. And then we're going to assign this newly created role. The scope in this case will be the main virtual network, we'll refer back to the role_definition that we just created, and we're going to assign the principal_id to the service_principal that we created in the previous step. Scrolling down a little bit more, we get to the actual peering process of the two virtual networks, and of course, you need to configure the peering on both virtual networks to have a successful connection. So the first resource creates the network peering connection on the main virtual network, so we're giving it the resource_group_name of the main virtual network, the virtual_network_name of that main virtual network, we're giving it the remote_virtual_network_id of the sec_vnet, which is why we needed the sec_vnet_id, and then for the provider, we're specifying that we want to use the azurerm.peering alias, so that will use the main subscription and the service principal to create this peering connection. You'll note that there is a depends_on at the end, and so far, we haven't seen this because Terraform is pretty good at figuring out dependency mappings within a configuration. For instance, if we scroll up a little bit, we have the role_definition and the role_assignment. I don't have to explicitly tell Terraform to create the role_definition first and then the role_assignment second because I'm referring to the role_definition within the role_assignment resource, so it just figures out, oh, well, I need

to create the role_definition first if I want to create this role_assigment later. Scrolling down and back into our network_eering configuration. In order for the service principal to successfully create this network peering, it needs to have permissions to create that peering from that role assignment, but that's not a connection that is obvious to Terraform. Because of that, we have to specify a depends_on of azurerm_role_assigment.vnet, which lets it know, wait for this role_assignment to be created before you try to create this peering connection. The second peering connection is from the security virtual network to the main virtual network, so we give it the resource_group_name of the sec_resource_group, the sec_vnet_name, the vnet_id of the main virtual network, and for the provider, we're specifying the security alias instead of the peering alias because of the subscription that we're working with is different, and once again, we're adding the depends_on because we don't want either of these peering connections to occur until that role_assigment is complete. So that's the entire VNET peering configuration. Now let's go ahead and add it to our existing configuration so that we can deploy it.

Deploying the Peering Configuration

Okay, we are in Cloud Shell, and we're logged in as Chris Jones. We can do an az account show to validate that we're using the correct subscription and that we're logged in with the correct user. So that all looks good, and we're in the terraform/1-main-vnet directory, which is where we had our existing configuration. If you destroyed the resources from the previous module, you're now going to have to recreate them by running terraform plan and apply. I did not delete those resources, so now I can just add the VNET peering configuration to this existing configuration. So the way that I'm going to do that is by running code vnet-perring.tf. That gives me the code editor. I'll go back to Visual Studio Code and select all and copy, go back to the Cloud Shell, and paste this in. There we go. Okay, and Ctrl+S to save and Ctrl+Q to quit. Now if we look at the directory, we now have this vnet- peering.tf, which will become part of our configuration. We also need to grab the values that were part of the next-steps.txt file from when Danny Brown configured the security VNET. So let's go back to Visual Studio, and we'll unhide the terminal, and this is the contents of the next- step.txt file. If you look at what we're doing, we're actually exporting a number of environment variables, and they all start with TF_VAR_ and then the name of a variable that's in our configuration. When you export an environment variable with TF_VAR_ and then variable name, Terraform will be able to find that environment variable and use it in the configuration. So let's go ahead and grab all of these commands, copy them, we'll go back to Cloud Shell, we're going to paste them in here, and now all of these values are stored in environment variables. The next step is to run terraform init just to make sure that we haven't added anything that isn't available from a provider, or a module, or backend perspective. That all looks good, so let's go ahead and clear the screen, and now we're going to run terraform plan. We have to specify one variable, which is the resource_group_name, and we're going to send this out to main.tfplan. So we'll go ahead and run this. Alright, and it's telling us that it's got four things to add. If we scroll up here, we can see that it's going to add a role_definition and a role_assignment. Scrolling down, we can see it's going to add two network_peering configurations, one for main and one for security. So that all looks good. Let's scroll back down to the bottom, and now we can run terraform apply main.tfplan, There we go, and it's going to go out, create that resource assignment, and then create those peering connections. So that resource assignment has already completed. That was very

quick. Alright, and it has finished successfully, so let's go ahead and minimize this Cloud Shell, and let's go into Virtual networks for our subscription as Chris Jones, and we can see that there is a vnet-main in here. That's the virtual network we've been working with. If we go into Access control, go to Role assignments, scrolling down, we can see that the vnet-peer service principal has been assigned the role allow-vnet-peer-main, which is what we were expecting. If we go down to Peerings within the settings for the virtual network, we can see that the main network is now connected to the security network, so we have successfully established a VNET peering between the main virtual network and the security virtual network using a service principal that was provisioned by Danny Brown. Our work here is complete. Excellent!

Summary

Alright, what did we cover in this module? Well, we covered a lot of stuff, so let's just take a step back and talk about it for a moment. We talked about using multiple Azure providers, both using AzureRM and Azure AD providers in the same configuration, like we did with Danny, and then also, using multiple instance of an AzureRM Provider, like we did with Chris. We had to use three instances of the AzureRM Provider, one for Chris and her subscription, one for the security subscription and the service principal, and another for the main subscription and the service principal, so that outlines very clearly why you might need to use multiple Azure Provider instances. We also saw how you can reference those aliases in resources and modules by using either a provider argument or a providers argument. Coming up in the next module, we are going to get into using Azure storage for remote state. We saw that a little bit with Danny's configuration. It was being stored in a remote state, but we didn't configure that remote state ourselves, so we're going to get into how you actually provision Azure Storage to do that, and then we're going to see how you can migrate state that's stored locally to a remote state configuration by adding a backend config. That's coming up in the next module.

Using Azure for Remote State

Introduction

Once you've gotten your feet wet with Terraform for a little bit, there's a pretty good chance you're going to want to move your locally stored state files to something in a remote location, and Azure storage is one such place where you can store those state files, and it actually supports a bunch of great features. So that's what we're going to talk about in this module. Hey, everyone, this is Ned Bellavance. I'm a Microsoft Azure MVP, and this is Using Azure for Remote State. All right, in this module, we are going to open with a brief discussion about Terraform remote state. If you've been using Terraform for a while, you're very familiar with remote state, but I do want to put a refresher in here so that I can highlight some of the features that are supported by Azure storage when you're using it for remote state. Now we'll talk about how to set up Azure storage and what some of your authentication options are if that is where your remote state is being stored. And then, finally, we are going to get into the

process of actually moving your state file from where it's stored locally to that remote location. But first, let's do a little refresher on what Terraform remote state is.

Remote State Refresher

There are basically two reasons that you might want to use remote state for the storage of your Terraform state file. The first is to safeguard the state file itself. If you are just storing your state file on your local system, then you are subjecting it to the whims of your local hardware, maybe that disk is going to fail, maybe you lose the workstation entirely. That's not a great scene if you're trying to manage your infrastructure using Terraform. So instead, moving that state file to a remote location that provides that additional redundancy is probably a good idea. It also enables teams to collaborate on shared infrastructure using a single configuration and a single state file, or in more complicated environments, multiple configurations and multiple state files. By putting the state file in a remote area, it enables special features that help with team collaboration. There are multiple supported backends for the storage of your state file, but not all backends are equal. There are some special features that only some of those backends support. One of those special features is locking. The whole idea behind locking is if someone is making changes or updates to the Terraform managed infrastructure, you don't want somebody else messing with it as well. So when something like plan, apply, or destroy is running, Terraform puts a lock on the state file saying, nobody else touch this. Something is going on. Another special feature is workspaces, and if you haven't come across workspaces before, it's basically multiple environments that are all using the same configuration, but have multiple state files, one for each environment that's being supported. How does Azure Storage stack up to these special features? Let's take a look at using Azure Storage for a remote state.

Azure Storage for Remote State

The storage you're going to be using in Azure for the remote state file is Azure Blob Storage, and Azure Blob Storage does support locking. So when Terraform is performing one of those operations, it's going to add an additional file that indicates that the state is locked. Azure Storage also supports workspaces. So when you create a new workspace and store data in the state file, it's going to create an additional state file that has the base key naming and append the workspace name to the end of that base key naming. So that's how it supports workspaces. Azure Storage also has multiple authentication methods that are supported by the AzureRM backend. So let's take a look at what those authentication methods are now. There are essentially four authentication methods that can be used with the AzureRM backend in Terraform. The first one is the managed service identity, and you would indicate that by adding the argument use_msi to the configuration block. That tells Terraform that it's running on an Azure virtual machine that has a managed service identity, and it can used that to access the storage account. The next option is using a shared access signature token, or SAS token, and you indicate that by adding a sas-token argument to the configuration block. What's nice about SAS tokens is you control the operations that are allowed by that token, and you can set a date for when that token expires. Something that you can't set a date for expiration on is the storage access key, and the storage access key for a storage account can do pretty much anything on that storage account. That's why it's

generally not recommended to use the storage access key as an authentication method; however, if you do want to use it, you simply put the access_key argument in the configuration block. Lastly, you can use a service principal for authentication. So assuming that you've configured the service principal with the correct permissions for the storage account, you simply have to supply the client_id in the configuration block, and you'll also need to supply the client_secret as well. Now let's take a look at how Globomantics is going to deploy the storage accounts for their remote state.

Azure Storage Setup

As we saw in the previous module, the security team is already using Azure Storage for remote state, but the storage account was actually created for them as part of the Terraform marketplace item. Now, Chris Jones wants to get in on the action, so she is going to set up her own storage account to house the remote state. So what she's going to need to do is deploy that storage account and then create a container within that storage account to hold the JSON file that is the remote state. So basically, if you happen to open the state file, you'll see it's actually JSON on the inside. In order to provide access to this container, Chris is going to allocate a SAS token when she's standing up this whole configuration and then share that out with those who will be running Terraform with this remote state location. So let's go over to the demonstration environment and help Chris get this deployed.

Deploying Azure Storage for Remote State

Okay, we are going to be using the Azure Cloud Shell to deploy this configuration, but first, let's review it in Visual Studio Code. Now you see, I have the holder 4-remote-state-prep open, and within there, there is a main.tf file I have open in the main pane. So let's go ahead and shrink up the EXPLORER to give us a little more room, and let's see what's in this configuration. It's relatively straightforward. We've got three variables here, one to name the resource group, one for the location of the resource group and storage account, and one for a naming_prefix that will be added to the storage account name. Let's scroll down a little bit more, and we get into the PROVIDERS. We're simply creating an azurerm provider and letting the Azure CLI supply the remainder of the settings. Scrolling down some more, we're using a new resource that we haven't seen before. That's the random_integer resource, and this is in the same family as the random_password resource. What you can do is give it a minimum and maximum in the arguments, and it will produce an integer that is within that range. We are going to be using this resource to help us with the naming process of the storage account to assure that it is globally unique. Scrolling down a little bit more, we create our resource group that will hold the storage account, and then finally, we are creating the storage account itself. And you'll see for the naming process, we're using a lower function for the naming prefix to ensure that the name is all lowercase because storage accounts don't allow for uppercase, and then we're appending that random integer using the result attribute of that resource. So at the end, we should have a name that resembles ITMA and the random integer. That should be globally unique for our storage account. We're putting it in the resource group that we just created, using the location that was defined in the variable, using the standard account_tier, this doesn't have to be a premium storage account by any stretch of the imagination, and for our purposes, we're going to use locally replicated storage. Of

course, you could use GRS if that was required in your situation. Scrolling down a little bit more, we are creating the storage container that will hold our Terraform state, and we're just naming it terraform-state to keep it simple and consistent. Then we also have to tell it which storage account in which to create this container. That's all we need to create for the actual storage of the state file. If we want to access that storage, we do need to create a SAS token. And rather than a resource, the SAS token is actually a data source, which sounds a little weird. But when you think about it, you're passing in a storage account, and you're asking for data about that storage account, and it sort of creates that token on the fly for you. So if we scroll down a little bit, what it needs in this data source is the connection_string for the storage account, which is one of the attributes of the storage_account primary_connection_string. You can set https_only to true, which we do want to use encrypted traffic, so of course, we're going to set that to true. In the resource_types, you're defining what types of resources this token can interact with. In our case, we've set all three resource_types to true. Scrolling down a little bit more, we define what services this token can interact with on the storage account, and we only need Blob, so blob is the only one set to true. The start defines when the token becomes valid, and we're going to use the function timestamp to give it the current time, and for the expiry argument, we're using the timadd function to add about two years of time to the current time, so this token will expire in two years. Now you could set this to whatever you want or actually use a variable to define how quickly you want this token to expire. Then we get into the permissions for the token, and this will apply to the Blob service. We're giving it read, write, delete, list, add, and create. So basically, it has full rights to manipulate the contents of what's in the Blob Storage for the storage account. So now, we have our token. In order to use this storage account and the token, we're probably going to want to render this information out to a text file. So scrolling down a little bit more, we get into a PROVISIONERS section where we're using the null_resource to run a local script. And the local script is going to echo out a number of values to a text file called backend- config.txt, so we're giving it the storage account name, the container_name, a key name, which will be the name of the state file when it's created, and finally, the sas_token. And as we'll see a little bit later, these are the exact key-value pairs that you need to configure the backend resource in Terraform. Scrolling down a little bit more, as OUTPUT, we have the storage_account_name and the resource_group_name. So that's the entire configuration. Let's go ahead and get this deployed in Cloud Shell. So we'll go over to the browser, and we're already logged in as Chris Jones, so we can go ahead and launch the Cloud Shell, and we'll go ahead and maximize the Cloud Shell to give ourselves plenty of room. Now let's go into the Terraform Directory, and we'll create a new directory for our remote state storage configuration. We'll go into that directory, and now we are going to create that main.tf file using the built-in code editor. Okay, let's jump back to Visual Studio Code, we'll select the entire configuration and copy it, go back to the browser, paste it into the code editor, do Ctrl+S to save and Ctrl+Q to quit. Alright, so our configuration is all set and ready to go. Now we have to run terraform init, and that's downloaded our plugins for the various providers that we're using, random, azurerm, and null. Let's go ahead and clear the screen, and now we're going to run terraform plan and pass it the one variable that we need, resource_group_name, and set it equal to itma-state, and then we'll send the plan file out to state.tfplan. We'll go ahead and run that, and it's going to add five new resources. That sounds good, so let me clear the screen here, and we'll do terraform apply state.tfplan. [00:12: 42.888] Alright, so it's going to go out and create these resources. It should go relatively quickly. Storage accounts usually don't take too long to create. Okay, there we go, and we can see, as Outputs, the resource_group_name as we set is

itma-state, and the strage_account_name ended up being itma72502, so thankfully, that was globally unique. Let's go ahead and clear this out and take a look at what's in our directory. We now have this backend- config.txt file that was created by that local exec provisioner. If we take a look at what is in that file, we can see it's got the storage_account_name, the container_name, the key name, and most importantly, it's got that sas_token at the end. We have successfully set up the storage account that we're going to use for remote state. Now how are we going to get that local state file that we already have in Cloud Shell up to this remote-state location? By migrating it, of course.

Migrating Remote State

Migrating Terraform state is actually relatively straightforward. The first thing you need to do is update the backend configuration with where you want the state file to reside now. How has it changed from the way that it was configured before? The next step is to run the terraform init process. If you think about what the init command actually does, it configures the backend, it downloads modules, and it downloads plugins. Now since it's configuring the backend in the initialization routine, that's what you need to run to update that backend to a new one. And then finally, when you run the initialization operation and the backend has changed, it will ask you to confirm that you want to change the backend and ask you if you want to move the existing state to this new location. In our case, we're going to say yes. Now what does a backend configuration look like? When you're creating a backend configuration, you start with a terraform configuration block because this is something specific to the Terraform configuration. Within that configuration block, you add a nested configuration block for the backend, and it starts with the keyword backend and then the name of the new backend type that you want to use. In our case, that's azurerm. And then within the nested backend configuration block, you can specify information about the backend, such as the storage_account_name. Now you see, I have the name of the storage account here hard coded as opposed to using a variable, and there's a good reason for that. You can't use variables in the configuration of your backend because the values in the backend are evaluated as part of the initialization process, which happens before Terraform evaluates the variables, so it can't actually use variables in the backend config. Now maybe storage account is something you want to change on the fly, but container_name might be something that you actually want statically defined within your backend, as well as the key. Those might be standard values that you use across all of your Terraform backend configs. If you want to dynamically set the storage account name or other properties during the initialization process, you can use a flag that's called backend-config and then supply it with a key-value pair for whatever setting you want to change. So let's say we didn't want to include the SAS token in our backend config, we wanted to submit it at runtime. We could use terraform init and use the backend flag with the key-value pair sas_token equals whatever that token is, and it will submit that value at runtime. this is something that is called partial config in Terraform parlance. So let's go back to the demonstration environment and get our backend configuration ready.

Implementing Remote State Migration

Alright, we're back in Visual Studio Code, and let me expand the EXPLORER for a moment. I have this directory, 5-remote-state. We'll expand that, and this has the backend.tf config in it. Now what's in there? Not much. We're simply defining a Terraform configuration block and then specifying the backend within that Terraform configuration block and telling Terraform it's going to be using azurerm. The rest of the configuration for this backend is going to be submitted at runtime using that backend config flag. So let's go over to Cloud Shell, which I still have running here, and we're going to go up a directory and go into our main-vnet directory. Whoops! And that has the state file for our currently deployed environment. Now if you've wiped out the environment from the previous modules, you can go ahead and run through the process of deploying it again now and get back to this point where you have the local Terraform state file. I'm going to assume that you haven't torn down that environment, and we're just going to add in the backend configuration. So let's launch the built-in code editor and create that backend.tf file. Okay, let's go over to Visual Studio Code. We're going to select this configuration, copy it, go back to the browser, paste that in, Ctrl+S to save and Ctrl+Q to quit. And now, if we take a look at the directory, we now have that backend.tf file. Alright, so we've got that configured. The next thing to do is we're going to copy the backend- config.txt file from that remote state folder to this folder so that we can use it when we run terraform init. So now we've copied that backend- config.txt file. We take a look at the look at the contents, we can see, there it is, backend- config.txt. That's all set. And if we take a look at what's in that backend- config.txt, it's basically four key-value pairs, storage_account_name, container_name, key, and the sas_token. That is a valid set of values that Terraform will look for when it's going to configure the AzureRM remote backend. Now we can run terraform init, and we'll specify the backend-config flag, and we'll set that equal to the text file where we have all of these key-value pairs, backend- config.txt, and we'll go ahead and run this command. Now it's going to ask us, do we want to copy our existing state to this new backend? If we do, we can simply enter yes. If we don't, we can say no, and it'll actually just create an empty state in the new backend as opposed to copying your values over. We do want to keep this configuration, so I'm going to enter yes. And there we go! You can see it says it has successfully configured the backend azurerm. So that is the new backend it is using. If we go ahead and minimize cloud Shell real quick and refresh our resource groups, we'll see there is this new itma-state resource group that should have our storage account in it. There is the storage account that we created. We can go into there, and we can scroll down to the containers portion of the Blob service, and there is our terraform-state container. And if we go into that, we can see there is a terraform.tfstate file in that container. We can click on that and click on Edit blob just to see the contents, and now we can see that this is in fact the state file that we're using to configure our main virtual network. We have successfully migrated our state from a local location to a remote location using Azure Storage.

Summary

Okay, what were the key highlights of this module? Well, the first thing that I do want to mention is that, generally speaking, unless you're doing something purely for dev, you should probably be using remote state by default. Get into the habit of using it. Azure Storage makes it super simple, and Azure Storage also supports locking and workspaces, which means as

your configuration grows and other people join it, you'll have these features already available to you. Now if you don't do it by default, we saw that migration is actually super simple. You simply add a backend configuration with the necessary values, and Terraform gracefully moves the state from its local location up to whatever new backend you have set. coming up in the next module, we are going to get into automating the deployment of our configurations, and the engine that we're going to use to drive that automation is going to be Azure DevOps using both repos and pipelines. That's coming up in the next module.

Using Azure DevOps

Introduction

As you're progressing through your adventure in infrastructure as code and adopting Microsoft Azure, there is going to come a point where you want to automate the deployment of infrastructure and integrate it into the application deployment lifecycle that the rest of your organization is using. And once you reach that point, you can start using tools, such as source control and CI/CD pipelines. That's what we're going to be talking about in this module. Hey everyone, this is Ned Bellavance. I'm a Microsoft Azure MVP, and in this module, we're using Azure DevOps. Alright, how are we going to approach our adoption of Azure DevOps? First, I'd like to start with infrastructure as code fundamentals. There are some fundamental things to think about when you're adopting infrastructure as code. Obviously, part of it is building templates, but there is more to it than just doing that. Once we've gone over the fundamentals, we'll dig a little deeper into source control and automation and how those can help you achieve better reliable and consistent deployments in your Azure environment. And then to help us put those principles into practice, we are going to adopt Azure DevOps as our deployment platform, [00:01: 17.124] so we're staying very much in the Microsoft Azure family. Now let's talk a little bit about IaC fundamentals.

Infrastructure as Code Fundamentals

When you think about infrastructure as code, I'm sure a few things spring to mind. It might just be the template system that you use, whether it's ARM templates, or Terraform, or Ansible, or some other tool. Basically, what all those things have in common is that they're software defined. You are defining the structure of your infrastructure through some sort of code. It's software defined. This isn't a manual process. Ideally, you'll apply some software development principles and make that software defined infrastructure both reusable in the form of modules or templates and repeatable, meaning that you can deploy that configuration in multiple environments. Because it's code, you're going to want to put it in source control, and I'll talk about that in more detail in a moment. Once you have a reusable and repeatable configuration, deploying it through automation just kind of makes sense. It's the next logical step, so we'll take a look at how you can do that as well. First, let's talk about source control management. As an IT Ops person, I know that I came to source control management a little bit late in the game, and that's because I was busy configuring switches and racking servers. I didn't really concern myself with the fact that I might want to store my source code somewhere that's, you know, safer than my desktop, but eventually, I did get

it. And in that process, I learned a few things. There are multiple formats when it comes to source control management. The most popular is probably Git, but you may have also encountered Team Foundation Version Control, which used to be SourceSafe back in the day. So if you were a Microsoft-heavy shop, you probably encounter it as SourceSafe. There's also Subversion, which you may encounter in some open source shops. There are also multiple platforms on which you can run this type of source control management, some of which are managed for you and some which you deploy and manage yourself. Some common examples would be GitHub, BitBucket, GitLab, and of course, Azure DevOps with their repos service. Finally, source control management is meant to enable two very important things. First of all, it's meant to enable collaboration. You can have multiple developers working on the same repository of code without stepping on each other's toes. It's also version controlled. So if you roll out a change in your code and it happens to break something, it's very easy to roll back to the previous version and then fix whatever the problem is, and that's especially important as your code rolls from development all the way through to production. Now how does your code get there? That is the job of CI/CD pipelines. CI/CD pipelines, or continuous integration and continuous delivery pipelines, are able to run on multiple different platforms, and some of the more common ones you might have heard of are Jenkins, Azure DevOps, and Atlassian's Bamboo. But there is a whole plethora of platforms out there that you could potentially run your pipelines on. What they all have in common is some sort of continuous integration for code checking. So when you commit your code as a developer, it goes into a continuous integration process and gets integrated with the existing repository of code and tested to make sure it's not going to break anything and that it merges successfully. Typically, it also builds or compiles that code into a releasable package. Now in the case of infrastructure as code, there is nothing to really build because you're not compiling source code. You're simply submitting a template and some variables. So that gets us into the continuous delivery portion of things. Generally, that's done in release pipelines. In a release pipeline, you are delivering the artifacts that are generated by the continuous integration process. And in our case, that's going to be the infrastructure as code configurations that we've put together being deployed onto a target environment. As part of this CI and CD process, ideally, you have automated testing and validation embedded in the process. So for instance, when you upload a Terraform script, maybe it's deployed in a development environment and a series of infrastructure tests are run to validate that the environment that's deployed matches what's intended. You might also validate the configuration to make sure that the formatting and syntax is correct. Once we've moved to the continuous delivery phase of things, generally speaking, the delivery moves through multiple environments, usually starting in a development environment and then moving to something like user acceptance testing, quality assurance, and eventually rolls into production. The movement of a release from one environment to another can be automatic, but oftentimes, there is a gate once you get to production where someone has to manually pull the lever to actually release that code to production. Terraform has an interesting way to deal with these multiple environments, and that is through the form of Terraform workspaces, so let's talk about those for a brief moment. Terraform workspaces are a way to deal with multiple environments. It all starts with a common configuration. You're using the same Terraform configuration files for each environment. What's different is that each environment gets its own individual state file, and that state file is basically what represents each environment. As you shift from one workspace to another, you're basically selecting a different state file to manage that other environment. So in that way, Terraform workspaces is able to support multiple environments from a single common configuration. It

also makes the current workspace available through our resource address, terraform.workspace. So within your configurations, you can use the value that's stored in terraform.workspace to make decisions about how the environment is configured. For instance, you may want different size virtual machines, depending on whether the environment is a development environment versus a production environment. So with all of this in mind, let's see how Globomantics is planning to leverage source control, automation, and Terraform workspaces in their environment.

Adopting Azure DevOps at Globomantics

Globomantics has made the decision to move forward with their use of Terraform when deploying resources in Microsoft Azure. They'd also like to adopt source control and automation, and what better tool to use than Azure DevOps since they actually get it for free through some of their MSDN subscriptions. So they have asked Chris Jones to go ahead and set up an environment in Azure DevOps to support their initiative. So, Chris Jones is going to do a few things. First, she is going to update the existing configuration that we've been using to take advantage of Terraform workspaces, and support multiple environments, and then she's going to establish source control management using Azure DevOps repos to store this updated configuration and keep it versioned and controlled. She's also going to use that source control as the basis to set up a release pipeline for the deployment of infrastructure. Now, in addition to source control, there's also some variables and values that need to be supplied at deployment time. Some of those values are going to be stored in a tfvars file that's also stored in source control, but some of them are going to be stored in variables in the release pipeline, especially things like secrets and passwords. So what Chris is going to do is have this source control act as the beginning of the pipeline, and also add in any variables that need to be defined in that release pipeline. Once we've entered the release pipeline, the first thing to be deployed is a development environment, and we're going to use Terraform workspaces to create that development environment. Assuming that everything goes smoothly with development and the deployment doesn't throw any errors, the pipeline is going to move forward to deploy a user acceptance testing environment. If that also deploys without issue, the process will move forward to a gated portion of the pipeline, where someone who has authority can decide whether or not to release this build to the production environment. So that is what we're going to be building in our demonstration. Let's go over to the demo environment, and first, we're going to review what's in the updated configuration.

Reviewing the Updated Configuration

Okay, here we are in Visual Studio Code, and we are going to review the updated configuration that's going to take advantage of Azure DevOps and workspaces. So I have the folder 7-azure-DevOps-workspaces open, and we're going to look at the main.tf file first. So let me open that, and I will shrink up the EXPLORER to give us a little more room, and as usual, we start with the variables. Some of these are the same as our original configuration, but some things have shifted a little bit. We still need a resource_group_name and a location, but when we look at the vnet_cidr_range, instead of having a list of strings, there is now a map in there, and the map has keys that corresponds to different workspaces and then a value for each key, which is the cidr_range that should be used for that environment. Below that, we're

defining our subnet names, web, database, and app, so we're creating three subnets, but you'll notice we don't have a subnet_prefixes variable. That's something we're going to allocate dynamically a little bit further down. Scrolling down, we get to our locals block. Locals is how you can define local variables within a Terraform configuration. It's equivalent to variables in an ARM template. And here, we're defining one local variable, the full_rg_name, or full resource group name, and this is a combination of the current Terraform workspace, which we referenced by doing terraform.workspace, and the resource_group_name variable that's submitted at runtime. So for instance, if this was the development workspace and the resource_group_name was vnet, then we would have a resource_group_name of development-vnet. Scrolling down some more, we get into providers. There's only one provider in this case, the azurerm provider, and all the values are going to be supplied by Azure DevOps at build time. Scrolling down further, we get into our first resource, and this is actually a data source called template_file. What the template_file data source allows you to do is create one or more strings based off of a template. In our case, we're trying to create the subnet prefixes based off of the virtual network IP address range and the number of subnet names that we have. So we set the count to the length of the variable subnet_names. In our case, we have three subnet names, so the count is going to be 3. Then we can either define a template in line or point it to a file. In this case, we're defining the template in line, and what we're doing is invoking cidrsubnet, which is a function. We're handing it the IP address range of the virtual network. We're adding 8 bits to the subnet mask, so currently, it's 16. We would add 8 and make it 24, so the subnet mask would now be 24 bit. And based off of the current count, we're selecting which subnet out of that range we'd want. We also have to give some variables for the template to use. And in this case, the vnet_cidr is being defined by the vnet_cidr_range variable, which is a map, and we're asking for the value that is at the key of terraform.workspace. If our workspace is development, let me scroll up to where we define our vnet_cidr_range map, the development key maps to the 10.2 .0 .0 /16 IP address range. Scrolling back down, we are going to set the current count for the template to count.index, so it's going to iterate through 0, 1, 2 because we have 3 subnets in our variable subnet names. This is a fairly common pattern for getting subnet prefixes based off of a larger block of IP addresses. It also means all we have to do is add another subnet name to our subnet_name variable, and it will dynamically create another subnet prefix for us. Scrolling down some more, we get into the creation of our Azure resource group. This is the same as it was before. The main difference now is the environment tag corresponds to the current Terraform workspace. Scrolling down some more, the last thing under resources is the vnet module from the Terraform Registry, and again, most of the settings here are the same, but I do want to point out a few things that are different. The address space is based off that vnet_cidr_range map, and we're using terraform.workspace as the key to get the value stored for that address space. Second, for the subnet_prefixes we are referencing our template file data source, and the way that we do that is data.template_file .subnet_prefixes, and then we have the square bracket with a star. Because we have multiple subnet prefixes defined by the template, star says, give me a list of all of those, and then the attribute I want from those is rendered, so I want the rendered text from each of those subnet prefixes. Terraform will take those results and submit it as a list to subnet prefixes. Then we have subnet_names. That's the same. And then finally, under tags, again, we're using terraform.workspace to define our environment tag. Scrolling down some more, we have our outputs, and these are just standard outputs. We're getting the vnet_id, the vnet_name, and the resource_group_name, and the reason we have these as output is in a later module, we are going to reference these outputs when we're deploying an application. So that's the entire

main.tf. Let's see what else is in this configuration. Well, we have our vnet-peering, and in this case, the vnet-peering has not changed significantly. We still need all the values relating to the security subscription, the virtual network, and the service principal that was created for us by Danny Brown. The changes here are mostly about the role definition and the vnet-peering resources, so let's scroll down to those and take a look at the role_definition. The main difference here is that the name for the role_definition adds terraform.workspace to the end, so if this was the development environment, it would be allow-vnet-peer-action-development. This prevents the roles from colliding with each other as you create a role for each workspace. Everything else about the role definition stays the same as it was before. Scrolling down some more, we get to the azure_role_assignment. That is the same. We're assigning the role we've defined to the service principal that was created by Danny Brown for us to do the network_peering. And then scrolling down some more, we are creating our two network_peering resources, one going from the virtual network we just created to the security network and the other one going from the security network back to this virtual network. And for both, we're updating the naming to use terraform.workspace as part of the name so we know which workspace this peering connection belongs to. So that is the vnet-peering portion of our configuration. Let's see what else we have in here. The next thing that we have is our backend.tf. And for that, we're simply defining that the backend is going to be azurerm. And when we create our pipeline, we will help define how it's going to use AzureRM for storage of the remote state. Now if you remember from the slide, we need some values to submit as part of this configuration, so I do have a terraform.tfvars file that defines a bunch of values to use for the variables in the configuration. And these can be pretty safely hard coded because the security subscription and the service principal are unlikely to change, as well as the virtual network and the virtual network id that we're peering to for security. One thing that might change is the service principal password, so we're not going to hard code that here, but we'll define that as a released variable. Also, because it is a sensitive piece of data, we might not want to check the password into source control. There is one more file in the directory, the workspacetest.sh, and we'll get to that as we set up our release pipelines. Now let's go over to Azure DevOps and review the source control configuration.

Using Azure DevOps Repos

Here we are logged into Azure DevOps as Chris Jones, and a basic building block of Azure DevOps is the project. A project holds within it things like your repositories and your pipelines. I already have a project created here called globomantics-testing. So let's go ahead and open up this project and see what's inside. As I mentioned earlier, we're going to be using Azure DevOps repos and Azure DevOps pipelines to form the source control management and CI/CV pipelines for our solution. Let's go into Repos and see what's defined in there. Within Repos, under Files, there is already a globomantics-testing repository, and within that, we have a networking folder. Expanding the networking folder, we can see that the file structure here maps to the file structure that was in Visual Studio Code, so I've already taken the liberty of adding these files to the globomantics-testing source repository. If we wanted to make changes to this repository, we could click on Clone and clone this to Visual Studio Code, make changes, commit them, and then sync them to this repository. We can also make changes in line. So if we wanted to change something about the main.tf file, we could simply click on the main.tf file, Azure DevOps repos would load the content, and by clicking on Edit, we can edit

something about this, so let's just add something in the comments here. We'll just say, Hello Pluralsight Learners, and then click on Commit. It will prepopulate the Comment field just letting us know that we've updated our main.tf file. You always want to add some sort of comment when you are committing something to Git. Once we've clicked on Commit, that has now been committed to the master branch, and it's part of our source code. We could look at the history of our commits and compare previous versions to what's in the main.tf file now. Now let's take what we have stored in Azure DevOps repos and deploy it using Azure DevOps pipelines.

Using Azure DevOps Pipelines

Based off of what we have in Azure DevOps repos, we can create a release pipeline for deploying this infrastructure in Azure. So let's go to Pipelines and select Releases. Like I said before, the build pipeline is more about compiling software and preparing artifacts for release. Because we're doing infrastructure as code, there's not really anything to compile in our case, so we can go straight to the Releases pipeline, and I have one pipeline defined here, workspaces. Let's go ahead and click on Edit and take a look at what's in workspaces. In the release pipeline, we start with artifacts. Where is the information coming from for this pipeline? In our case, We're using Azure DevOps repos as our artifacts for deployment. And if we look at this little lightning bolt here, there is an option to enable continuous deployment. We won't set that up right now, but what that means is every time a Git push happens to the selected repository, it will kick off a new release to deploy the updated source to the environments that are in the stages portion of the pipeline. Right now, we don't have that turned on, but we will turn that on later so you can see how it works. Go ahead and close that out. The next thing you'll note is that we have a development stage in our Stages area, and that development stage has a lightning bolt as well. So if we click on that, it will take a look at what triggers this portion of the stage. And right now, the trigger is set to After release. So if a release is created either manually or through the continuous deployment, it will kick off this development stage. Let's go ahead and close this out and take a look at what is in the development stage. It has 1 job and 5 tasks. Here are the tasks that are set up to run on an agent. Now let's look at the agent first before we look at the tasks. The agent is using a hosted agent that's provided by Azure DevOps. Basically, it spins up an Ubuntu 1604 agent and runs these tasks on that agent. When the tasks are complete, the agent is shut down and discarded, so you don't have to do anything to set up a persistent agent of any kind. You absolutely can do that, but for our purposes, it's not necessary, and it's actually a little bit more work. On our agent, we are going to run a number of tasks. Now because this agent is being spun up from a base configuration, it does not have Terraform preinstalled. so the first thing that we need to run is a Terraform installer process. This task and other Terraform tasks are available from the Azure DevOps marketplace. If we click on plus here to add a task, it will ask us what task we want to add. And if we start typing in Terraform, you'll see it has two tasks that we've already selected to run on the agent, but there are additional tasks that exist in the marketplace. And if you wanted to use one of those, you could highlight it and click on Get it for free and it will add that item to your marketplace. We don't need to actually add a task at this point, so I'm going to close that out, and we can take a closer look at what's going on in this Use Terraform 0.12 .10. That's the display name, and the only thing that we need to specify in the Terraform installer is what version of Terraform we want it to install. In this case, we want to install 0.12 .10 to keep it consistent with the version we've been using for all

of the other deployments. Once Terraform is installed successfully, we can go through the terraform initialization process, so let's select that task, and you can see it's running terraform init as its command, and there is a Configuration Directory defined. let's scroll this over a little bit to give more room for the Configuration Directory. And as you can see, it's using the system.default working directory as its base directory for the configuration, and what Azure DevOps actually does is it clones down the repository that's in Azure DevOps repos to this agent so that you can use the files that are in source control. So what we're doing is running init in the networking directory that has all of our files that make up our configuration. And if we scroll down a little bit more, you can see it provides you the ability to specify a backend yourself. In this case, we want to use azurerm. Then we have to specify the details of the backend configuration, so let's go ahead and drop that down. And we can see that it's asking for the Resource Group Name where our storage account exists, the Storage Account Name, which is the storage account that we created in the previous module, 72502, the Container Name, which is terraform-state, and then finally, what's the key we want to use for the storage of the state file? In this case, we're using terraform.tfstate. Now remember, as we use workspaces, the workspace name is going to be appended to terraform.tfstate, so we won't have a collision of keys based off of different environments. We're going to be okay on that one. Speaking of workspaces, the next task in the process is a Bash Script, and the Bash Script is intended to check whether or not a workspace exists and create it if it does not. Let's take a look at that script in VS Code. Now I told you we were going to get back to that workspacetest script, and this is the script right here. It's very straightforward. Basically, what the script is doing is running terraform_workspace_list to get the list of current workspaces and then running grep with the -c argument, which gives us the count of how many times it finds a search term. We are going to supply the search term as an argument to this script, say development. So if in terraform workspace list, it does not find development, if it equals 0, then this will test it to true and the script will run terraform workspace new and then the name of the workspace that we want to create. If it does find the workspace, this will test to false, and we'll get to the else statement. The else statement in this case will run terraform workspace select and the name of the workspace. So basically, we'll be selecting to use the development workspace if it already exists. That's the entirety of what this script does. Let's go back to the task in pipelines. Now we can't see what the Script Path is set to at the moment, so let's click on the breadcrumbs, and we'll see that the location is globomantics-network/networking/workspacetest. That is where our script is. That's correct. The next thing is the Arguments. We need to pass it a workspace name. The argument that we're using is Release.EnvironmentName. That is a built-in variable in pipelines that gets you the name of the stage that you're currently in. And as we can see, the stage that we're in is development, so it's going to submit development as the argument. If we create another stage called, say, uat, then it would submit uat as the argument. Expanding the Advanced for a moment, we do have to set the Working Directory as well. Where should this script be run from? And the path for the working directory is the same as the script path. We want it to run in the same folder as all of our other configuration files. Once we've either created the workspace or selected it, we can move on to terraform plan where we plan out the deployment of our resources. So we'll go ahead and select terraform plan here, and you can see the Command it's running is plan, the Configuration Directory is networking, that's where our .tf files are, and then we get to select an Azure subscription where we want to deploy these resources. We're telling it we want to deploy these resources in the PS subscription. Scrolling down some more, we do have some Command Options here, and the Command Options specify a variable inline for the sec_client_secret, which is

the service principal password, and that we want to store the plan in main.tfplan. Now the way of referencing a variable in pipelines is by doing that dollar sign and parentheses and then putting the name of the variable in there. The variables are stored on the Variables tab, so let's go over to the Variables tab, and we can see we have one variable defined, sec_client_secret. And if we scroll over a little bit and look at this little lockbox, it says that it is currently locked, so pipelines knows to treat this as secret text, and it won't show the variable in pain text in the logs. It'll keep it all just stars. So that is the one variable we have defined in our pipeline variables. Now let's go back to Tasks. At this point, we have run terraform init, we have selected our workspace or created it, and we've created the terraform plan for what we're going to deploy. The last thing to do is run terraform apply, and you can see, we are funning the apply command. If I can line this out a little bit, our Configuration Directory is the same as the one we used for init and plan, we are using the same Azure subscription that we used before, and the only command option that we're submitting as part of our apply command is the main.tfplan file, so this is the plan file that we generated with terraform plan. Assuming that each task completes successfully, the agent will run the next task in succession. At the end of all the tasks, we should have a successfully deployed virtual network that is peered with the security network. In order to kick off this process, all we have to do is create a release.

Deploying the Development Workspace

Now that we've reviewed the configuration of our pipeline, it's time to create a release. And for that, all we have to do is click on this fun button that says Create release. And what it will do is trigger a release and kick off the development stage. Let's go ahead and click on Create here, and that release has now been created. We can click on the hyperlink here to go to Release 12, and this shows us the same pipeline, but in the view of a release happening, and it's currently set to Queued. So we can click on that, and basically, it waiting for an agent to be ready to accept the job. Once it's accepted, it will go through the tasks that we've defined for this agent. The first thing it does is download the artifacts, and then it's going to install Terraform as we asked it to do. Now it's running terraform init to initialize our Terraform configuration. Now it's running the Bash script. That was very quick, and now it's running terraform plan to plan out what's going to be deployed in our environment. It looks like that succeeded, so now we're going to the terraform apply portion of the task where it's going to create that virtual network, the role definition, and the VNET all as part of the apply process. So let me scroll down a little bit and wait for the console output to complete. Alright, and all of the tasks have completed successfully. That infrastructure should be in place. Let's go over to the Azure Portal, and we'll take a look at our Resource groups, and in there, you can see there is now a development-vnet resource group. And if we click on that, we should see that there is a virtual network that is part of this resource group. There it is, development-vnet. We can click on that and take a look at the Peerings that are configured here and see that we have a connected peering between this virtual network and the security virtual network, so our deployment has been successful. Now let's go back to Azure DevOps and add the uat environment to our stages.

Adding the UAT Workspace

Alright, we're back in Azure DevOps, and we're going to add a uat stage to our existing pipeline, so let's go back to the workspaces pipeline, and we are going to select to edit the workspaces pipeline. Now as you can see, we only have one stage defined. It's development. Now what's nice is that you can actually clone a stage, like this, and then click on that stage, and all we have to do is rename the stage uat and then click Save up here, and click OK, and that's all there is to it. Now you can see there's a little lightning bolt here with pre-deployment conditions. And if we click on that, by default, it is going to trigger this UAT stage if the development stage is successful. That's what we want it to do. If we've successfully deployed to development, we would like it to move on and deploy to uat. So let's go ahead and close this out, and we are going to create another release here, and you can see now in the pipeline, it's going from development to uat. We'll go ahead and click on Create, and now it has created a new release for us. We can go ahead into this release and see that currently development is Queued. We can go ahead and click on that and see that it is preparing an agent for the job. Now there shouldn't be any changes to our development environment because we haven't made any changes to the source code. It should just roll through and say that no changes are required. Now since this will take a little while, I'm going to speed up the video so we don't have to wait for the entire deployment of the development stage. Okay, our development stage has succeeded successfully. So now we can go back to release 13 and see that our uat stage is queued up. That's the next thing that is going to deploy. The Agent job is about to start, and it is provisioning an agent to run this deployment. This one's going to take a little bit longer than the development because it doesn't have any of the resources yet. It has to create these resources. Now what I want to say while this is deploying is while you're seeing green now, usually when you first set up a pipeline, it's going to be a lot of red. Your releases are going to fail, and the reason is because there are going to be assumptions you've made or things that don't work quite like you expected them to. That's normal, and don't get discouraged. When I was first setting up this demonstration, I had about 13 failed releases before I finally got a green release, and that felt great! Once it goes green, it tends to stay green because you've worked out the kinks of your pipeline, and now you have an automated way to deploy your infrastructure. Alright, our uat stage has also successfully deployed. If we go back to the Azure Portal and take a look at the Resource groups, we can see that there is now a uat-vnet resource group, within there, there is a uat virtual network, and if we look at the Peerings for this virtual network, we can see it is also peered with security, so the configuration was successful in this regard as well. If we take a look at our storage accounts to see what's happening with remote state, here is our storage account, 72502, and let's go into Containers, and there is the terraform-state container, and within that container, we're going to see there are three files now. There is our original terraform.tfstate before we started using workspaces, and then there is the terraform.tfstateenv :development and uat. So you can see how it appends the name of the workspace to the terraform.tfstate key. Now you remember, one of the things that I talked about was enabling continuous deployment, so let's go ahead and enable that now.

Adding the Production Workspace

Looking back at our original diagram, the idea was to set up continuous deployment from development to UAT and then to production, but to have a gate for approvals before it

hit production. So let's go back to Azure DevOps, we'll set up the production stage, enable the gate, and enable continuous deployment. Alright, here we are in our pipeline configuration, and we're ready to add the production stage to our environment, and all we have to do is click on Clone for the uat, and then click on the stage itself, and change the Stage name to production, and click on Save. Now remember, we want this to be gated. We don't want it to automatically deploy. If we look at the lightning bolt here, it's going to deploy this as soon as uat is done, but we want to have a gatekeeper here, so we can click on the person icon here, and scrolling down, we can require Pre-deployment approvals. And under Approvers, we can select our self as an approver. So basically, once it gets to this stage, it will pause here until Chris Jones comes in and actually approves this portion of the deployment. That all looks good, so we are going to click on Save here, and hit OK, scroll up, and we can exit out of this. The other thing that I talked about setting up was continuous deployment, so we can go to this lightning bolt here and set up a continuous deployment trigger, we will set this to enabled, so every time we commit to our repository, a new release is going to be created. Let's go ahead and click on Save on that and exit out of here. So now we have continuous deployment set up and three stages in our pipeline. Let's go into Repos and make a change to one of the files. Let's go into Networking, let's say, and go into main. Now let's say we wanted to add a tag to our resource group deployment. Let's go ahead and click on Edit here, and then we'll scroll down to where we defined our resource group, and let's add another line, we'll call it costcenter, set it equal to it, and go ahead and click on commit to commit this change to source control. We'll click on Commit here to commit it with the comment. Now that that change has been committed, let's go back to our Releases pipeline, and you'll see that an additional release has been created, Release-14. This was automatically created because of the commit. We can go ahead into this release and see that it's currently in the development stage. Now the only change here is the tag, so this shouldn't take very long to deploy to development and uat. So I'm going to speed up the recording so we can get to where it hits the production gate. Okay, it has successfully gotten through the development and uat stages, and now it's gotten to the production stage, and it's waiting on our approval. It's not going to deploy to production until we've approved it. Let's go ahead and click on the Approve button here, and we could leave a comment saying Update tag and go ahead and click on Approve again. Now that this has been approved for production deployment, if we close out of here, you can see that it has queued a job for production deployment, and it will go through the same deployment process. Now because we haven't deployed to production yet, it will be standing up net new infrastructure, so this will take a little while, so I'm going to speed up the recording. Okay, and here's that red that I talked about. It looks like something went wrong with production, and I wonder what it was. We can go ahead and click on Failed, and it actually will give us the error as Terraform was going through its process. Now it looks like what it's saying is that the key production was not found in vnet_cidr_range. Is that correct? Let's go ahead and close this, and we can take a look at our repositories. We'll go into networking, into main.tf, and let's take a look at what we have in here, and it's absolutely correct. The map says prod and not production, so we can go ahead and edit in here, change the variable to production, commit it, and click on commit again, and guess what's going to happen? It is now going to create a new release and go through the Release pipeline to fix the mistake. So let's go back to Pipelines and go into Releases, and now Release-15 has kicked off based off the change that we made. Let's go into Release-15, and we can see how the stages progress to production. And obviously, I'm going to speed this up so we don't have to wait around for each stage to complete. Alright, and we're back at Pending approval for our production deployment, so we'll click on Approve and

say that map has been fixed, which hopefully it has, and click on Approve here, and now it will move forward with the production component of the deployment, and we can go into the queue for this and actually watch the deployment process, and now it's running the job on our agent, so I'm going to speed up the recording again because this is going to take a little while to deploy all the resources for the production environment. Okay, and this time, our production deployment succeeded. If we look at the entire Release pipeline, we've got green across the board. That's really nice to see. If we go back to the Azure Portal and go into our Resource groups, there is now a production-vnet resource group, and as you might assume, in there, there is a virtual network, a production-vnet. And within that production-vnet, if we go down to Peerings, we can see that the production network has also been peered with the security network, so we have accomplished the goal of setting up Azure DevOps to deploy our infrastructure as code in an automated fashion.

Summary

Alright, that was a heck of an adventure. Let's sum up what we've learned in this module. Well, first thing that I want to stress is please, please use source control. I'm using source control for all of the example files for this course. Anything that you're building that you think you might want to share or at least just keep track of, I would recommend using source control. There's a ton of free options, and there's even private repositories in GitHub, so you don't have to make it publicly available. Also, automate your deployments. First of all, you're going to learn a lot about your deployments just through the automation process. Assumptions that you might make when you're doing something manually will get called out when you try to automate that thing, and that helps sharpen your skills as an engineer. Also, we didn't really have time for it in this module, but I highly recommend adding testing for your infrastructure. There's a number of different packages out there for testing, and there's actually one specifically for Terraform itself called TerraTest, which Microsoft uses. So if that seems to be something you're interested in, I'd recommend searching out TerraTest and checking it out. Coming up in the next module, it's time to deploy an actual application on top of this infrastructure. So we are going to build upon this infrastructure and also access the existing state file as a data source for the next configuration, and we'll also see how you can deploy an ARM template using Terraform, which sounds a little funky, but it actually comes up more often than you might think. That's all coming up in the next module.

Using Data Sources and ARM Templates

Introduction

After you deploy your first Terraform configuration, there's a pretty good chance you're going to deploy additional configurations that have some dependencies or need some information about that original configuration. You might even say they're using it as a data source. There's also a chance that you already have some things deployed in Azure that you're using ARM templates to maintain, and you may not be quite ready to refactor those templates, so that's what we're going to be talking about in this module. Hey everyone, this is Ned Bellavance. I'm

a Microsoft Azure MVP, and this is Using Data Sources and ARM templates. Alright, what are we going to get into in this module? Well, like I said in the opening, we're going to be talking about what I like to call layering configurations, the idea that you are going to have multiple deployments of multiple configurations in your environments, and some of those configurations are going to need information about each other. There is a layering and a dependency that exists between different components and deployments. The way in which that information is consumed is often done through data sources, and that data source could be an Azure Provider data source, or it could actually be the remote state of that other configuration that you're consuming as a data source, so we'll see how you can incorporate either of those scenarios into a configuration. And then finally, as promised, we are going to talk about ARM templates and how you can use them in your Terraform configurations and why you would use them in your Terraform configurations. But first, let's talk a little bit about layering configurations.

Layering Configuration and Using Data Sources

Now I want you to imagine in your mind that you have an existing configuration deployed in an Azure subscription, say a virtual network. Now what are you going to be putting in that virtual network? Because it's not very useful all by itself, you are probably going to be placing some sort of application in there. Let's say you have a business logic app, so now you have one deployment, that is your networking, and a second deployment, that is your business logic app. Your business logic application is going to need to know information about that virtual networking environment to be able to deploy successfully. Let's add another component to this environment. Let's say you're using an app service. Well, the app service may have a dependency on the business logic app that may consume information or interact with that logic app, so it may need to pull information about that business logic app, but it doesn't necessarily need to know about the virtual network, just the business logic app itself. You may also be deploying Azure SQL in your environment, and the app service is consuming that. Now you could, of course, put the app service and Azure SQL in a single Terraform configuration and deployment together, but what if you're using Azure SQL for more than just App Service? Do you deploy Azure SQL as a separate configuration, as a module within the App Service configuration, or some other way entirely? Let's look at another potential scenario. You have a database warehouse service in your virtual network, so when you deploy that, you need information about the virtual network. In addition, that database warehouse may be publicly accessible through a DNS entry. And in this case, we're using Azure DNS for the deployment, so you're going to need an Azure DNS zone to already exist in order for the database warehouse to be able to successfully add a record to that DNS zone. So do you create the DNS zone as part of your database warehouse deployment? Probably not. That zone is probably going to be used for other services as well, but the database warehouse does need to be able to query Azure DNS for that zone in order to add a record successfully. So these are just some of the different ways that configurations can become layered and interdependent. In some cases, you'll be pulling this information from Azure as a data source. So what are some of the more common data sources that exist in Azure you might use? Well, one we've already seen a few times, and that's pulling information about the subscriptions that you have access to in the environment. That's actually been fairly useful for configuring roles or even setting up additional AzureRM providers. You can also pull networking information, and that might be a

virtual network, it might be a subnet configuration, it might be a load balancer. There's a bunch of different data sources that exist within the Azure Provider so you can get information about the networking. You can also query about custom images. You may have a build process where custom images are created and stored in a gallery in Azure, and you use those images for your virtual machine deployments. You may want to be able to query that gallery to determine what the newest version of an image is and use that image in your VM deployment. Another one that is commonly used is key vaults. Key vaults are a secrets management and lifecycle solution within Microsoft Azure. You can store keys, secrets, and certificates in it. Those are all very useful things for other deployments, say if you're spinning up an app service and it needs a certificate or a secret, it may want to use Key Vault as a data source. And finally, recovery vaults allow you to query existing query vaults to automatically configure backup for your Azure virtual machine, and that seems fairly useful. You're probably not going to create a new recovery vault for every virtual machine you deploy, so instead, you can query for information about that recovery vault to properly set up backup. Now how would you actually configure some of these data sources? Let's look at a few examples. If you wanted to query for all the subscriptions available for the currently used service principal, you could use the AzureRM subscriptions data source, and you'd do that by simply using the data keyword followed by azurerm_subscriptions for the data source type and then give it a name that you can refer to in the remainder of the configuration, perhaps subs. If you wanted to use Key Vault as part of your configuration, you would, again, use the data keyword, the data source name is azurerm_key_vault, and then give it a name. You also have to provide some information about that key vault so it can retrieve the proper one, so you have to specify the name of the key vault, as well as the resource group name that key vault resides in. Finally, for the AzureRM virtual network, it's very similar to the key vault. You have to give it the name of the virtual network and the resource group name that houses that virtual network, and then you now have access to all of the information about that virtual network. It's a very useful data source as you're trying to configure Azure virtual machines or load balancers. Now let's take a look at how Globomantics is planning to use some of these data sources.

Provisioning the Application Remote State

So far in Globomantics, we've worked with Chris Jones to set up a basic virtual networking environment for the deployment of applications. Now it's time to actually start laying down some of those applications, so we are going to be working with Hector Sanchez, and we are going to be deploying an application within the main VNET that is in each environment. What does Hector want to deploy? He'd like to deploy an app tier that consists of two virtual machines and also a front-end load balancer for that app tier, and this is an internal-only app, so the load balancer is not going to be deployed with a public IP address. It all resides within the app subnet within the main-vnet. As part of setting up this configuration, Hector is going to make use of remote state storage, so we are going to have to set up a place to store that app remote state. Why don't we go ahead and take care of that now in the demonstration environment? Okay, we're here in Visual Studio Code, and I've got the folder 8-app-remote-state open. We are going to open the main.tf file, and I'll go ahead and minimize the EXPLORER to give ourselves a little more room. This is basically the same configuration that we used to deploy the storage account for remote state storage for the networking infrastructure. So I'm not going to spend too much time going through the configuration. We're basically specifying a resource_group_name, a location, and the

naming_prefix for our storage account, we're using the azurerm provider, we're creating a random_integer for the naming of the storage account, we're creating a resource_group to house the storage account, we're creating a storage account, and within that storage account, we're creating a storage container named terraform-state to hold that state. Scrolling down a little bit more, we're creating a SAS token to access the storage account to be able to write the remote state and read it. And if we scroll down to the PROVISIONERS, we are going to be using the local-exec provisioner to create a file, backend-config.txt, that has all of the key-value pairs we're going to need in order to properly set up our remote state backend. Now one slight difference between this configuration and the previous one is we're going to be running this locally as Hector Sanchez, and Hector is running from a PowerShell prompt. So I've updated the local-exec provisioner to use PowerShell commands instead of Bash commands to send the proper information to the backend- config.txt file. So it's a small difference, but it does illustrate how you can use PowerShell or Bash in your local-exec provisioner. So let's go ahead and open up the terminal window, and we're already in the proper folder. I just want to confirm that I am logged in as Hector Sanchez, so let's run az account show, and we can see I am logged in as Hector Sanchez using the PS subscription. So I just wanted to make sure before I ran any other commands that everything was set up properly. Now we can go ahead and run terraform init to initialize the local backend we're using to provision this storage account, as well as the provider plugins. Okay, that was successful. Now we can run the terraform plan, and I actually have the exact command stored in this commands.txt just to make it a little bit easier for you and me. So we're going to run terraform plan and pass the resource_group_name of itma-app-state and the naming prefix of appitma. So let's go ahead and grab this whole command here, copy it, and we'll paste it down here, hit Enter. So this should go out and plan to create the resource group, as well as the storage account and give us that SAS token. Okay, that has completed its planning successfully, so we're going to go ahead and run terraform apply state.tfplan, and now that's going to go out and provision that storage account, and it should write the information that we want out to that backend-config.txt file. Okay, good. It completed successfully. If we look in our file explorer again, we can see there is now a backend- config.txt file. If we open that up, we can see it has the storage_account_name, container_name, the key, and the sas_token we'll use to write to the storage account, so it looks like everything was set up correctly for the app state remote storage. There is one missing component, and that is how Hector Sanchez is going to get the information about the networking environment.

Adding the Networking Remote State

In addition to setting up a storage account for the storage of the remote state for the application deployment, Hector is also going to need access to the network state for the network configuration, and that is because he is going to use information that is stored in that networking state to access the information about the main-vnet in the deployment. Let's take a look at how that is going to be configured. In order to use a remote state as a data source, we're going to use the data source type terraform_remote_state, and of course, we have to give it a name, and in this case, networking probably makes the most sense. And then within the data source, you have to specify what type of backend is holding the remote state. In our case, we're using azurerm. And then within a configuration block, you have to specify information about where that remote state is configured, as well as how to access

it. So in our case, we're using an Azure Storage account, so we have to specify the storage_account_name, the container_name, the key that's being used, and in our case, we're going to be using a SAS token to access that storage account. So let's go back to the demo environment and provision that SAS token to access the networking storage account. Okay, here we are logged into the Azure Portal as Hector Sanchez, and you can see there is an appitma storage account and an itma storage account. The itma storage account is the one that has the networking remote state stored in it, so that's the one that Hector needs access to to get information about the networking state. So we'll go ahead and click on that storage account, and we want to provision a SAS token. So we'll go down to Shared access signature. And for Allowed services, we only need access to the Blob services, so we'll uncheck all the other ones. We do want to allow the resource types of Service, Container, and Object, so we'll leave those selected. Now in terms of permissions, Hector only needs Read and List. He doesn't need any other access to this storage account, so we'll remove these levels of access so Hector can't accidentally do something to the network state and corrupt it. We'll leave the Start date as is and set an End date, let's say, a month in the future, and then go ahead and generate the SAS token. Now that that's been generated, we'll go ahead and copy that SAS token to the clipboard, and we're going to use that SAS token in our configuration to access the network state. Where is that information being stored? Let's go over to Visual Studio Code, and in Visual Studio Code, I have the folder 9-app-deploy open. This is where the configuration resides for the application that Hector Sanchez wants to deploy. In our terraform.tfvars file, we have a network_state variable being defined here, and it's a map type that has a number of key-value pairs. One of these key-value pairs is sts, which is the SAS token, so we can go ahead and paste the SAS token we just grabbed into that portion of the variable. The storage account we also need the name of. So if we go back to the portal, the storage account is itma72502. So we'll just put in 72502 here, and now the storage account is correct in our tfvars file. The container name and the key name are already configured correctly, so we don't have to do anything there. We can just go ahead and save this file. Now let's go ahead and take a look at the actual main.tf file to see what we're deploying in our configuration.

Reviewing the Application Configuration

Now that we have access to the networking remote state all worked out, let's take a look at what's actually in our main.tf file. And I'll go ahead and minimize the EXPLORER to give us a little more room here. For the backend, we are going to be using azurerm to store our remote state, so that's set up correctly. For our variables, we will be defining a number of variables here. We'll need a resource_group_name for where we're going to deploy the application, a naming_prefix for the application, scrolling down some more, we're setting the location as a default of eastus, and then we have our network_state variable that we just defined in terraform.tfvars. So this is a map of strings, and within the description, we're defining that we need sa, cn, key, and sts as keys with the appropriate values assigned. Below that, we're specifying a vm_count for our variable, so how many virtual machines do we want to provision? Hector 1 and 2 so we have the default set to 2, and then we have a local variable being defined, and that local variable takes the terraform.workspace and then adds a dash and the naming prefix, and we'll use that for naming some of the resources within the configuration. For instance, if we were in the development workspace, the prefix would be development-itma. Scrolling down some more, for PROVIDERS, we're using the azurerm

provider, and we're using the Azure CLI for authentication, so we don't have to put anything in there. Then we get down into our data sources. So here, we have the actual data sources that we'll be using. The first data source is an Azure data source. We're getting the current subscription. The second data source is our terraform_remote_state for networking. And just like we saw in the example earlier, we're specifying the backend as azurerm and then specifying the configuration. Each of the values for the configuration are stored as a key in the variable network state map. So for storage_account_name, we're specifying the variable network_state and the key sa, same thing for container_name. The keys is a little bit different because we have to get the key name, but we are going to be using workspaces with this. If we go back to the portal very briefly and go into the Containers for the networking remote state, go into terraform-state, we can see that the naming of the keys is terraform.tfstateenv : and then the name of the workspace, so we have to make sure that our key that we're using as a data source matches that format. Going back to Visual Studio Code, we can see that we've set up key to be whatever we specify in the network state map and then appending env: and the current terraform.workspace. And then finally, the sas_token that we set up, which is a read-only SAS token for this storage account. So that's going to set up our Terraform remote state. Scrolling down a little bit more, we want to deploy this application on the app subnet within the virtual network, and so we are using the data source type azurerm_subnet to query information about that app subnet. In order to do that, we have to specify the virtual network name. We're getting that network name from the terraform_remote_state data source. One of the outputs that was part of that configuration was an output named vnet_name, and now we're actually consuming that output. So in order to get any information out of configuration, that information has to be exposed via outputs within that Terraform configuration. Likewise, the resource_group_name is also an output of that Terraform configuration, so it's available from the remote state as resource_group_name, so that's how we're querying for our AzureRM subnet. Going down into RESOURCES, we'll first provision a resource group using the prefix that we created in the local variables and appending -app to it, and then we're going to create an availability set to store our virtual machines using the prefix for the naming and adding -aset to it. Scrolling down some more, we get into the network_interface, so each of those virtual machines will need a network interface. We use the count argument to specify the number of network interfaces we need, which should be equal to the number of virtual machines, so it's set to var.vm_count. The naming will be the prefix variable, dash the index of the count, -nic. And if we look down in the ip_configuration, we're using the azurerm_subnet data source to tell it which subnet_id to put this network interface in, so that is how we're using our azurerm_subnet data source. Scrolling down some more, we're provisioning our Azure load balancer. For naming, we're going to use the prefix for the name of the Azure load balancer. And just like the network interfaces, our front-end IP configuration for the load balancer is also going to be on this subnet, so we're using the same data source, data.azurerm_subnet .app .id. Scrolling down a little bit more, we have to establish a backend_address_pool that the virtual machines will participate in, and then we have to create an address_pool association between the network interfaces that we created further up and the backend_address_pool that we just created. It's a little confusing if you've never worked with Azure load balancers before, but basically, we need to create a backend_address_pool and then associate the virtual machines, or more specifically, the NICs for those virtual machines with that backend_address_pool so the load balancer has something to send traffic to. Within that configuration, we need to create an association for each network interface, so we're setting the count argument to vm_count, the

network_interface_id we can get from the resource azurerm_network_interface.app and specifying the count.index with the square brackets, which will return the appropriate network interface from the list of network interfaces that we created earlier. And then for the backend_address_pool_id, we're just giving it the id of the backend_address_pool that we created directly above it. Now it's time to create the virtual machines, so we're setting the count to the vm_count. That's the number of virtual machines we want to create. For the naming of those virtual machines, we're using the prefix and then adding - and the count.index, so the first virtual machine will be the prefix -0, and then we're specifying the location, the resource_group_name. For the network_interface_ids, we're using the same syntax that we used for that backend_pool_association using azurerm_network_interface.app and then using the square brackets and then which element out of the list of network interfaces we'd want for this configuration.id. We're associating the virtual machines with the availability_set we created earlier. The size of the virtual machines will be a DS1_v2. We're also setting the operating system disk and data disks to be deleted on termination. For the storage_image_reference, we're using Ubuntu 16.04 for our storage_image. For the storage_os_disk, we're using a name that corresponds to the virtual machine name. For the os_profile, we're specifying the computer_name, which is a combination of the naming_prefix, the count.index, and then VM. The admin_username will be tfadmin, and I'm giving it a very simple admin password. You can change this if you feel like you need to. We're not actually going to be logging into these servers anyway. And then finally, for the Linux config, we are setting disable_password_authentication to false because we are specifying a password. So that's everything in the configuration. And if we scroll down to OUTPUTS, the only output here is the load balancer private IP. And since we're specifying this as an output, that would enable a separate Terraform configuration to use the remote state from this configuration and get that private IP address if it needed it. That is the configuration, so let's go ahead and prepare for deployment.

Deploying the Application Configuration

We've completed reviewing the configuration, so now, let's get started on deploying it. Let's go ahead and open up the terminal, and you can see we are in the directory 9-app-deploy. The first thing we need to do is copy the backend-config from the previous deployment so we can use it to initialize our remote backend for this deployment. So now we have that backend- config.txt file in this directory, we can run terraform init --backend-config and set it equal to the backend- config.txt file. And as you can see, it successfully initialized the backend and also downloaded the provider plugin for the azurerm provider, and now we're going to create a workspace to deploy the application configuration. So we'll do terraform workspace new development, and now we're using the development workspace. So we're going to be deploying this application into a development workspace, and it's going to be using the development Vnet for its deployment. Now we can run terraform plan -out app.tfplan. And the reason we can do that is because we have all of the additional values stored in the terraform.tfvars file in this same directory. I'll go ahead and run that. And as you can see, it is going through the process of refreshing state for all of the data sources, and that includes, not only the Azure-based data sources, but that remote state data source, and there we go. It looks like it was able to successfully access the remote data sources, including the remote state, to be able to properly plan out the deployment. So let's go ahead and run terraform apply app.tfplan, and while that's deploying, let's go back to the

portal, and we'll go back home and go into Resource groups. And within Resource groups, we can see there is a development-itma-app and a development-vnet. If we go into development-vnet, that has our development-vnet virtual network. We can go in there. And if we look at the connected devices, we can see there is now a load balancer and two network interfaces, and all of them have been provisioned in the app subnet, so that's what we were looking to do. We wanted to provision two virtual machines and a load balancer in the app subnet, and we were able to do that by querying for information from both Azure and the remote state. If we go back to the Resource groups and take a look in this development-itma-app resource group, I'll just shrink that up a little bit, we can see that it has successfully deployed the load balancer, the virtual machines, the availability set, the disks, and the network interfaces. So all the components have been deployed. And if we go back to Visual Studio Code, we can see that the deployment has completed successfully, and the output is that private IP address of the load balancer. So we are all set. We have deployed the first application in our networking environment. But wait! There is more to deploy. Let's talk about ARM templates for a minute.

ARM Templates with Terraform

This whole time, we've been talking about using Terraform to replace ARM templates, so now it might seem a little strange that we're bringing ARM templates back into the mix, but there's a few reasons why that might happen. The first reason is you simply have existing templates that you're not ready to refactor, maybe somebody else wrote it, maybe you wrote it, and you just don't have the time. So if you have those existing templates, you can still use Terraform to manage your deployments, but use the ARM template to actually do the deployments, and we'll see how that works in a moment. Another reason is because the Azure providers are updated as often as they can be, but there may be times where a particular resource is not yet supported by the AzureRM Provider or the Azure AD Provider, but it is supported through an ARM template. During that period of time, you could use an ARM template as a stop-gap measure until the provider gets updated. Now the way that you use Terraform to deploy ARM templates is through a resource, a template resource, so let's take a look at how that template resource is used. Just like any other resource you might deploy with Terraform, it all starts with a resource configuration block. Now the resource in question here is the azurerm_template_deployment resource, and we have to give it a name, so we'll call this one template. The first argument is name, and this is the name that you'll see if you go into a resource group and look at the deployments. Each deployment has a name. This is the name that is going to appear for that resource group deployment. You also need to specify a resource group where this deployment is going to happen. That's no big surprise if you've used ARM templates for a while, and then you have to specify the actual template itself, so there's really two ways to do this. You can specify the entire template body in line using heredoc syntax, but that's not very appealing to the eye. The easier and more common option is to use the file function and just point it at the JSON file where your ARM template already exists. Now in addition to having a template file, you do need to specify parameters for that ARM template, and that is handled by the parameters configuration block where you'll just have a series of keys and values that correspond to the parameters that are required by your template. And finally, you can specify a deployment_mode for your template. By default, it's set to Incremental, but you can set it to Complete mode. Now let's see how Globomantics is planning to use the ARM template deployment in their application deployment architecture.

Adding the Subnet Delegation

So far, Hector has been responsible for the application deployment, but now he is going to need a little bit of help from Chris Jones. They're going to work together to get this additional component of the application deployed. So let's take a look at our existing architecture that we have so far. We've got our main virtual network, and in there, we've already deployed the first application. It's this app tier with an internal load balancer. Now it's time to add an additional component to the application. This is going to be the web front end that talks to this app tier, and that's going to run in Azure App Service. In order for the App Service to access that internal load balancer, it's going to need to talk to this virtual network, and the way that you do that in Azure is by creating a subnet delegation. And what this is is a dedicated subnet within the virtual network that is associated with the App Service. So it can go in through this subnet delegation and talk to the app tier through that internal load balancer. Now since Hector doesn't own the networking component of the deployment, he is going to ask Chris to set up the subnet delegation for him in her current configuration of the networking, so let's go ahead and set that up now. So here we are in VS Code, and I have the folder 10-ARM-template open, and within there, there is a subnet_delegation.tf file. Let's go ahead and open that and take a look at it. And basically, all this is doing is creating an additional subnet called appservice. This resource is going to be added to the existing network configuration that's already been deployed in development UAT and production. Because of that, we can reference resources that exist in that configuration. For the virtual network, we're referring to the virtual network that's created by the vnet module. For the address_prefix, we're simply looking at the existing number of subnets and grabbing the next available address space after those subnets. Then we have to specify a delegation for this subnet, and that is done through a nested delegation block where we specify a name for the delegation, we're calling this appservicedelegation, and then which services should be delegated. The name for the service delegation is Microsoft.Web /serverFarms, and then there are three actions that needs to go into that delegation, which is the virtualNetworks/subnets/preparteNetworkPolicies/action, virtualNetworks/subnets/action, and virtualNetworks/subnets/join/action. If this look like it's a role, it's because it basically is. We're granting the Microsoft.Web /serverFarms permissions to perform these actions on this subnet. So we're going to take this configuration and add it to the existing configuration that's already been deployed for the networking. So we'll go ahead and grab this entire configuration and copy it, and let's go over to Azure DevOps where we have our configuration being deployed, and here we are in Azure DevOps, we're logged in as Chris Jones, and we are in the repos where our networking configuration is stored. We can go into the networking folder, and now we can click the drop-down and do New, File. We'll call the file subnet-delegation and click on Create. Now it will allow us to paste in contents for this file. There we go. We've pasted in the contents, and we can click on Commit. It will allow us to add a comment for the commit. Since we have continuous deployment already set up in this pipeline, it's going to create a new deployment. And if we go over to Pipelines and into Releases, we can see that a new Release, Release number 21, has been kicked off. We can click on Release-21 and see that it's currently in the development state. If you skipped the module for Azure DevOps, then you're going to need to go ahead and deploy the networking environment using at least the development workspace and add in this subnet-delegation.tf file to make sure that subnet delegation has been created within the development-vnet. While this is progressing through the other environments, let's go over to the Azure Portal and take a look at our resource groups. This should be updating the virtual

network that's in the development-vnet resource group, we'll go into there, and now we'll go into the development-vnet virtual network. And if we take a look at the subnets here, we can see there is now an appservice subnet, and we can see it has been delegated to Microsoft.Web /serverFarms, so the development environment has been updated successfully. If we go back to Azure DevOps, we can see it has succeeded on uat. And if we would like to roll this change into production, we can click Approve and Approve here, and the change will be rolled into the production environment as well. So we've successfully added our subnet delegation, and now we can move forward with the application deployment on app services.

Deploying the ARM Template Configuration

The App Service application deployment is based off of an existing ARM template that was written by somebody else. Now unfortunately, Hector doesn't have time to refactor this ARM template to a Terraform configuration. He'd like to just deploy it as is for now and refactor it later. That's no problem. He can go ahead and use the template deployment resource to deploy this template. So let's go over to VS Code, take a look at the configuration, and get it deployed. Alright, here we are back in VS Code, and if you look in the folder 10-ARM-template, there is an azuredeploy.json file and a template- deploy.tf file. The azuredeploy.json is the ARM template, so we can go ahead and take a look at that first. And looking at the template, we have a number of parameters listed out here. It's got the webAppName, the sku of the App Service plan that we're going to use, the location where this will be deployed, scrolling down some more, we have the vnetName that contains the subnet we'll be delegating, and then the actual subnet reference itself, which is the id of that delegated subnet. If we scroll down some more, we get into the variables for the ARM template, and we're simply defining the webAppPortalName and the appServicePlanName. Then we get down into Resources, and there's not a whole lot of resources being deployed within this template. If we scroll down, we can see that the first thing that's being deployed is a server farm, which is basically an App Service plan. Within that App Service plan, we're going to provision the next resource, which is Microsoft.Web /sites, which is the actual App Service itself. And for that, we need to refer back to the serverFarmId of the server farm we just created and specify and explicit dependency. You'll remember, in ARM templates, you have to explicitly define dependencies, whereas in Terraform, you generally do not. Scrolling down some more, we get into a nested resource for this App Service, and this is where we defined the delegated subnet for this App Service to connect to. For that, the type is config, and the name is virtualNetwork, and we have to specify that this is dependent on the creation of the website itself first. And within the properties, we specify the subnetResourceId, and we also have to add this additional property, swiftSupported true. So that's everything in the ARM template. It's really just creating a simple app service in an App Service plan with this subnet delegation. Now let's take a look at the Terraform configuration. So we'll expand that out and open up template-deploy. There we go. For the template deployment, we need to know some information about the virtual network and subnet that's been delegated, and we can do that through a data source. So the data source is azurerm_subnet, and we know that the name of the subnet is appservice. The network name we can get from the networking remote state, as well as the resource_group_name for this subnet. Scrolling down some more, we first will create a resource group to hold this template deployment, and then we'll do the actual template

deployment itself. So you can see, the name that we've given it is weappdeployment, the resource_group_name corresponds to the resource_group we just created. For the template_body, we're specifying file and then the azuredeploy.json file we just looked at, and we do have to pass it the parameters that didn't have default values. So the webAppName will be the prefix local variable -web, the vnetName we can grab from the remote state, and the subnetRef we can grab from the azurerm_subnet data source that we created further up. And then finally, the deployment_mode is set to incremental. Alright, we're going to add this to existing application deployment, so we're going to have to copy these two files to the folder 9-app-deploy. Let's go ahead and do that now. So I am going to copy the azuredeploy file, and then I'll go ahead and copy the template-deploy file. Okay, so both of those files, if we expand 9-app-deploy, we can see that the azuredeploy file is there and the template-deploy file is there now. Let's go ahead and clear out the terminal, and we'll shrink this up and just bump this up a little bit so we can see the entire output as it's happening, and all we need to do is go ahead and run terraform plan, again. This will refresh the remote state from networking, so we'll get that additional App Service subnet that's been delegated, and we'll now grab the appservice subnet as its own data source as we can see, and it now knows it has to add two new resources, one is the template deployment and the other one is the resource group. So that looks all set. Let's go ahead and copy this, and we'll run terraform apply, and that will go out and create that resource group, as well as create that app service. Now there is one important thing to note about the template_deployment resource. It will create the deployment object, and then that deployment object will create resources within Microsoft Azure. If you destroy the template_deployment resource within Terraform, it will not destroy the resources that were created by that deployment. The workaround for that is for each deployment to create a resource group and then delete that resource group as part of the deletion of the template_deployment, and in that way, you're guaranteeing that anything that was created as part of the deployment is also destroyed when you destroyed that resource group. Alright, there we go! It has successfully deployed. Let's go over to the Azure Portal and take a look. Alright, here are the resource groups, and we can see there is now a development-itma-webapp that is for our template_deployment. Let's go into that, and we can see we have an App Service plan and an App Service. Let's go into the properties of the App Service. And if we scroll down on the side, going into Networking, there is the VNet Integration option. We can click here to configure. And if we scroll down a little bit, we can see that it's already connected to a VNet, the development-vnet, that's what we wanted, and under Subnet Details, we can see that the Subnet NAME is appservice. We have successfully deployed the ARM template and attached this App Service to the delegated subnet in the VNet. Good job everybody.

Summary

Alright, what are the key takeaways for this module? I want to highlight a few things here. One is when you're designing your Terraform configurations, it's important to consider the overall architecture of what you're planning to deploy and some of the interdependencies there. That will lead you to determine whether you should create a single monolithic configuration, should you break things into modules for reusability, or should you create several separate configurations and tie them together through a remote state and data sources. My advice would be to keep configurations relatively atomic, single units that can be deployed on their own. There may be some dependencies there, but you don't want to create

these monolithic configurations that will break easily when one single thing changes. It's better to try to take this more atomic and loosely coupled approach. The last key takeaway of this module is that ARM templates are an option. They're not a great option. Ideally, you would refactor that ARM template as a Terraform configuration. But if you don't have the time or that's not currently an option because of the providers, you can absolutely use ARM templates as a stop-gap measure.

Course Summary

Now that concludes the course. So let's just do a high-level summary of the course in general and what are some things that you could do next. At a high level, you've learned about the multiple providers from Microsoft Azure that exist in Terraform. That includes AzureRM, and Azure AD, and also, the Azure Stack Provider, which we didn't get into, but it's very similar to the AzureRM provider. We also went over the many ways that you can authenticate to these providers, whether it's through a service principal, the Azure CLI, or a managed service identity. We also saw how you can configure a backend to store state remotely using Azure Storage and then use that remote state as a data source for additional configuration. And then finally, we saw how you can use some of the native tools within Microsoft Azure and Azure DevOps to implement source control and automation. Now where can you go next with this skill set? Well, I'm sure there's a lot of places you can go with it. This probably opens up a few doors for you. But if you wanted to do some experimentation on your own, I would highly recommend going through the documentation for the providers and experimenting with other interesting resources. There was obviously not enough time in this course to cover the multitude of resources that exist in the provider, so my goal was to give you a background in the high-level generalities of the provider and allow a deep dive as an exercise for you. You could also get started on infrastructure testing. One of the things that we didn't get to in this course was how to properly test your infrastructure once Terraform has provisioned it. So if that seems interesting to you, I'd highly recommend checking out TerraTest and maybe even Pester to do some infrastructure testing in your automation pipeline. And that will conclude the course. I want to say thank you so much for taking the time to learn more about HashiCorp Terraform and Microsoft Azure. I'd love to hear your feedback about the course and ideas for additional course. You can find me on Twitter, it's @ned1313, or leave a comment in the Discussions tab of this course. Until next time, go build something great!