Advanced Terraform with Azure

Introduction

Course Overview

Hello, everyone. My name is Ned Bellavance, and welcome to my course, Advanced Terraform with Azure. I am a HashiCorp ambassador, a Microsoft MVP, and founder of Ned in the Cloud. As you may have guessed, if there are two things I'm passionate about, it's HashiCorps Terraform and Microsoft Azure. If you've slogged through using ARM templates or click opsing your way through Azure, get ready to leave that all behind. In this course, we are going to extend your Terraform skills when deploying and managing infrastructure on Microsoft Azure. Some of the major topics we cover will include authenticating with the Azure providers, importing existing Azure resources into Terraform code, using Terraform modules to build out common Azure infrastructure components, and automating your deployments with Azure DevOps. By the end of this course, you will have an in depth understanding of how Terraform can supercharge your Azure experience. Before beginning the course, you should be familiar with the basics of using Terraform at a foundational level and the common infrastructure building blocks of Azure like virtual networks and virtual machines. I hope you'll join me on this journey to learn more about integrating Terraform and Microsoft Azure with the advanced Terraform with Azure course, at Pluralsight.

Welcome to the Globomantics!

To help put this course in a larger real world context, you have just joined Globomantics as a DevOps engineer on their platform team. They're in the process of spinning up a new application code named Taco Wagon. The company has been using Microsoft Azure for a few years now and that continues to be their platform of choice. However, they're new to Terraform, and they've brought you on board to help out with adoption. You'll be responsible for accomplishing several goals around the Taco Wagon deployment, including bringing existing resources under Terraform management, deploying virtual networks to host the Taco Wagon components, deploying virtual machines, virtual machine scale sets, and Azure Kubernetes service clusters to host the application components, automating the deployment process with Azure DevOps, and dealing with Azure preview features through the AzAPI. Globomantics is excited to have you on board and is ready to step into the world of infrastructure as code and Terraform. You'll be able to test your skills with labs throughout the course, but you may also want to spin up your own Azure subscription to test out

modules and features that aren't supported by the lab environment. I'd also recommend installing an IDE like VS Code that has support for Terraform language plugins to enhance your coding experience. Lastly, this course also relies on the use of the Azure CLI, so you'll want to have that installed locally or use it through the Azure CloudShell. Now let's get started.

Setting the Stage with Azure

Setting the Stage (Introduction)

Before we get into using Terraform with Azure, there are a few things we need to do to set the stage. In this section, we are going to cover some key decision points and considerations for using Terraform with Azure. We will also cover some of the tools and techniques we will be using throughout the course. We will start by looking at the various options we have for running Terraform. There's my editor of choice, VS Code, but there are other options as well, including the Azure CloudShell. Then we will walk through the process of setting up VS Code for optimal Terraform and Azure integration. There are several extensions we will want to install and configure to make our lives easier. Terraform would be nothing without the providers that allow it to interact with the various cloud platforms. We will take a look at the Azure provider ecosystem, which is more than just the AzureRM provider. Of course, if you want to use one of these providers, you need to authenticate to Azure AD first. We'll check out what options are available when running locally in Azure or in an automation pipeline. Finally, you'll get the chance to put all this knowledge into practice with a hands on lab. You'll use Terraform to deploy a container group in an existing resource group in Azure.

Picking a Development Environment

In this lesson, we will look at some of the options available for working with Terraform in Azure. Picking a development environment is a personal choice, and while my personal preference is VS Code, you can use whatever you are most comfortable with. VS Code is a free, open source cross platform editor that is very popular with developers. It has a large ecosystem of extensions that can be used to extend its functionality. It probably also doesn't hurt that it's actively developed by Microsoft, so the Azure integration is pretty good and it's what you'll see in most of their tutorials and labs. Extensions in VS Code provide some additional functionality beyond what's built into the editor. There are thousands of extensions out there, but I wanted to focus on a few that are most useful for this course. The first one is the Terraform extension maintained by HashiCorp. This extension works with the language server to provide syntax highlighting, code completion, and other features. I find it indispensable

when working with Terraform. The next one is Azure resources, an extension maintained by Microsoft that provides an easy way to view resources in your Azure subscription. Microsoft also maintains the Azure Tools extension pack. It's actually a bundle of several extensions that help you work with Azure. I feel like this one is optional if you already have the Azure CLI installed. It does provide some nice built in actions and the ability to explore certain resources, but I don't use it much. Finally, there is the GitHub extension. While this isn't an Azure or Terraform thing, I tend to store most of my Terraform code in GitHub repositories, so I find this extension useful for interacting with those repositories. If you're using a different source control system, there are extensions for those as well. I also want to take note that there is an older Azure Terraform extension that is no longer maintained. It's still available in the marketplace, but I would recommend using the official Terraform extension instead. Let's head over to my demo environment and take a look at how to install these extensions.

Demo: Setting Up VS Code for Terraform Development

VS Code is my code editor of choice when working with Terraform, so let's walk through how to set up VS Code to work with Terraform and Microsoft Azure. We'll start by looking at VS Code and then installing some of the extensions that I mentioned earlier, and then we'll navigate around the IDE, deploy some code using Terraform, and see how the extensions can help out with the whole process. Let's head over to my demonstration environment and dig in. I'm starting with my local installation of VS Code where I've removed most of my usual extensions so we can walk through the process of installing them. I'm going to start by opening the Extensions tab and searching for the Terraform extension, and we're looking for the official HashiCorp one. I'll click on Install and move on to the next extension. Now, I'm going to search for the Azure resources extension, and I'll install that one as well. Once that's installed, I'll search for the Azure Tools extension pack and install that as well. That one's going to take a little while since it's actually a bundle of extensions, but once it finishes, finally, I'll search for the GitHub Repositories extension and we'll install that one too. Now that we have all of our extensions installed, let's take a little tour of the IDE. One of the things I really like about VS Code is the ability to pop up a terminal window and run commands right from the editor. I can do that by clicking on the Terminal menu and selecting New Terminal. This will open a new terminal window at the bottom of the screen. Alternatively, I can use the keyboard shortcut Ctrl+` to open and hide a terminal window. You can have multiple terminals open at the same time, in case you have a long running command or an SSH session in one and you want to run some commands in the other window. Another useful feature is the ability to split the Editor window. I can do that by clicking on the Split Editor right button. This will split the window horizontally. I can also split the window vertically by right clicking on the tab and selecting Split Up or Down. I can then open another file and drag that file down to the appropriate split where I want to

view it. When I'm working on multiple Terraform files, the ability to quickly compare them side by side is extremely useful. On the left, we have all the various extensions and some built in views. Included are things like the File Explorer view that we see now, the source control view. Once we have our local Git configured, we can use the integrated source control in VS Code to make commits and push changes to our repositories. I'm going to start a Terraform configuration using a Terraform block to define my required providers. As I type, the Terraform extension will provide suggestions to complete what I'm typing. It will also link the code showing where I have possible issues. For instance, I'll add a provider block to this configuration, and right now, it's showing that something's missing inside that provider block that's required, and in fact, for the AzureRM provider, we have to put in the features block for Terraform to process it correctly. Adding a new resource to the configuration, as I type AzureRM, it helpfully shows me all of the existing resources that are part of the AzureRM provider and narrows it down as I type more of the resource type. In this case, I'm looking for an AzureRM resource group. As I continue to fill out the resource, the red squigglies let me know that something isn't quite right about this resource yet. I'll add the name for the resource group called in my group, but I still have the red squigglies here. Let me hover over that and I can see that I have not specified the location argument yet and that is required for the resource group. Let's go ahead and add that as well. Once I've added the location argument, the angry red squigglies go away, and we can be assured that we've at least satisfied the required arguments for this resource type. Now, I'm going to quickly deploy this Azure resource group and then we can check out what the Azure Resource Manager extension can do. While I'm doing that, you might have noticed that I haven't been saving my file as I'm typing in it, and that's because I have the Auto Save feature enabled from the File menu. Now, personally, I really like this feature, but it's not for everyone. I'm going to launch the Azure extension, and first, it prompts me to sign into Azure to enable this extension. I'll click on the sign into Azure, and in a separate window, it has prompted me in a browser to sign in with the relevant account I want to use with this extension. I'll go ahead and do that, and once I sign in, the subscriptions I have access to will populate inside of the resources. Now, I blurred out some of these subscriptions because you don't need to see all of the subscriptions I have access to. The resource group that we've deployed is specifically in the MAS subscription, so I'll expand that one, and if we scroll down through all the resource groups in here, we should see one called my group, that's the resource group we just deployed, so we can get a visual confirmation on the deployment of resources that we deploy with Terraform. You can also view additional commands available in the extension by hitting F1 and starting to type Azure, and it will give you a list of commands you can run that leverage the Azure Tool extension. That's everything I want to show in VS Code. Now, we need to discuss the various providers that are available to work with Azure and Terraform.

Navigating the Azure Provider Ecosystem

While you might think that working with Azure means using just the AzureRM provider, there are actually a whole host of providers that interact with Azure in some regard, so let's dig into the Azure provider ecosystem and see what's out there. The main provider you'll be using for most Azure resources is the AzureRM provider. This is the official HashiCorp provider, meaning that it is maintained and supported by folks at HashiCorp. There's also a dedicated team at Microsoft that works on this provider. This makes it the de facto provider for working with most resources in Azure. Despite that, there are some limitations to the AzureRM provider. It's not always up to date with the latest Azure features and doesn't support all Azure resources. In particular, it doesn't support preview services or features that are in beta. The engineering resources at HashiCorp and Microsoft are limited, so they have to prioritize what they work on. This means that some features and resources will be added to the provider before others. To provide a stop gap measure, there is the AzAPI provider. This is a partner tier provider, meaning that it is supported and maintained by a dedicated team at Microsoft, and it's intended to act as a thin abstraction over the Azure API, hence the name. Anything you can configure with the Azure API, you can configure with the AzAPI provider. The downside is it's not as user friendly as the AzureRM provider. We'll take a look at how to use it later in this course. Another important provider in the Azure space is the Azure AD provider, and this is an official HashiCorp provider. This provider allows you to manage users, groups, and other resources in Azure AD, which is now called Entra ID. Now I wouldn't expect the name of the provider to change anytime soon since it's pretty hard to get everyone to update their existing code to support a new provider name, so for the time being, this is going to be the Azure AD provider, although the service is now called Entra ID. There's a few other related providers that we probably won't get into in this course, but are certainly worth mentioning. All of these providers are maintained by Microsoft. The first one is the azuredevops provider. This provider allows you to manage Azure DevOps resources, such as projects, repositories, and pipelines. If you're using Azure DevOps for your CI/CD pipelines, this provider can be pretty useful. Microsoft also maintains the azurecaf provider. This provider doesn't actually provision any resources in Azure, instead, it includes helpers for working with Azure landing zones, specifically a naming helper and naming convention helper. If you're using Azure landing zones, this provider can help you with achieving consistent naming conventions. Continuing with the theme of landing zones, there's an alz provider that has some native data sources that are not available in the AzureRM provider and gets around some limitations of the AzureRM provider. Regardless of which provider you use, you'll need to authenticate with Entra ID first. Let's take a look at how to do that.

Authenticating to Azure

When it comes to authenticating with the AzureRM provider and the Azure providers in general, there are a multitude of ways to do it largely depending on who is authenticating and where they are authenticating from. Let's take a look at some of the options. The first and most common option for people just getting started is the Azure CLI. Simply log in using the az login command and select which subscription you'd like to provision resources in. Terraform will detect that you're logged in with the Azure CLI and use that authentication if no other is specified. In the demonstration you saw earlier, you didn't see me have to authenticate when I deployed the resource group because I'd already authenticated using the Azure CLI. If you're running Terraform directly in Azure, you can use managed security identities, AKA MSIs, to authenticate. This is a great option for automation pipelines since you don't have to worry about storing credentials anywhere, but it does mean that you're limited to running on a compute resource that's in Azure. Another option is to create a service principal in Entra ID and authenticate with the service principal. You can use a client secret, which is akin to a password, a client certificate, or a federated credential. This is a good option if you're running Terraform from an automation platform and you don't want to use an MSI, but you still need to store the client secret or certificate somewhere secure. The federated credential is a little bit different. It uses OpenID Connect, which is often called OIDC to authenticate. The basic idea is that you establish a trust relationship between Entra ID and another identity provider, such as GitHub or HashiCorp Vault. That provider can issue a token to a client which will be passed along to Entra ID and verified by Entra ID. If that token is valid, Entra ID will issue a token that can be used to authenticate to Azure. This is a great option if you don't want to store credentials anywhere and you're not running in Azure. Why don't we put some of this theory to work in a hands on lab?

Building a Foundation (Azure Networking)

Building a Foundation

Before we get too far down the path of creating new and exciting resources on Azure with Terraform, first, we need to address two important things, storing state data and importing existing resources. We are going to start this section with a look at how you can leverage Azure storage to store your state data remotely. There's more to it than just creating a storage account and pointing your Terraform configuration at it. We'll explore the best practices for configuring the storage account, how to properly configure RBAC back to limit access, and options for authenticating and accessing the storage with Terraform. To reinforce what you just learned, there's a setting up remote state storage lab where you can create an Azure storage account and update an existing configuration to use the storage account as a state back end. Next, we have to acknowledge that

you probably have existing resources in Azure that you may want to start managing with Terraform. Having a green field deployment is the exception rather than the rule, so we are going to look at a declarative import process and a special tool developed by Microsoft that makes importing existing infrastructure a lot easier. We will use the declarative import process to start managing an existing virtual network with Terraform, and then you'll have a chance to use the Azure Export for Terraform tool in a lab to practice importing resources for yourself. With these two pieces in place, we'll be ready to start building new infrastructure in Azure and where better to start than with networking. So much of what you deploy on Azure requires a virtual network, and we'll take a look at the common components that make up a virtual network deployment and what should be managed as part of a network configuration versus elements you might want to deploy in a separate configuration. As part of our exploration, we'll take a look at some of the popular networking modules on the public registry. After all, there's no reason to re-invent the wheel. We can use one of the modules to create an application-specific virtual network and peer it to our shared services virtual network from earlier. Then you'll get a chance to try out the same scenario in a lab environment. There is going to be a lot of hands-on components in this section, so roll up your sleeves and let's get started.

Using Azure Storage for Remote State

When you first get started with Terraform, you'll probably just use the local file back end for state data storage. It's the default and requires no set up, but once you move to a collaborative environment or a production scenario, it's time to move that state data to a remote location. In this lesson, we will start by exploring how Terraform can leverage remote storage for state data. Then we'll take a look at how to properly configure an Azure storage account to house your state data. There's some pretty important information in there that you don't want just anyone to have access to. Finally, we'll cover how to migrate your existing state data to Azure storage. Turns out that's a pretty simple process. The biggest challenge is deciding how to authenticate. First, let's dig into remote state functionality. State data is Terraform's mapping of the configuration to a target environment. It's how Terraform knows what resources it's managing in Azure. When Terraform generates an execution plan, it is trying to make state match what's in the configuration and outlining the actions it needs to take to make that a reality. What I'm trying to say is that state data is critical to the proper functioning of Terraform, and you don't want to leave that state data just hanging out on your local workstation. Instead, you can define a state back end for Terraform to use for storing state data. There are a ton of available options, including AWSS3, Google Cloud Storage, Terraform Cloud, and of course, Azure Storage. And the state data is stored in JSON format, so really anything that can store JSON could be used for state data. In Azure, the state is

stored as a blob in object storage, so we need to set up a storage account and a container in the storage account for blob storage. State data can hold sensitive information, and it's also critical for the functionality of Terraform, so our two guideposts for configuring Azure Storage are durability and security. We'll start at the storage account level. Since we care about durability, you should select globally redundant storage as the storage type for your state data. If a region fails, you won't lose all of your state data. From a security perspective, you should require HTTPS for all operations and TLS 1.2 at a minimum. Depending on how you plan to authenticate, you can also disable storage key access, forcing all authentication to use Entra ID. On the networking side, you have a few options depending on your scenario. The most restrictive is disabling public access and using Private Link, but that assumes that you'll be running Terraform from a virtual network or an internal network that has access to the Private Link endpoint. If that's not the case, you can enable public access from selected virtual networks and IP addresses. Now this assumes that you know what virtual networks and public IP addresses your Terraform runs will be coming from. If that's not the case, you can opt for the most permissive option and enable public access from all networks. For data protection, you should definitely enable soft delete for both blobs and containers, and enable versioning for blobs and blob change logging. Although you can't easily roll back to a previous version of state data, it's nice to have tracking and versioning enabled for auditing and logging purposes. Encryption on storage accounts is enabled by default, and you can choose to use Microsoft managed keys or your own customer managed keys. Customer managed keys are preferred if you're okay with the additional administrative overhead. There's also an option to enable infrastructure encryption, which double encrypts your data. I'd recommend against this, unless there's a compliance requirement you need to adhere to. And that does it for the storage account configuration. Now on to the container configuration. There are a lot less options at the container level. Basically, you just want to ensure that anonymous access level is set to private. Once you've created your storage account, it's time to configure permissions. Storage accounts can use Entra ID or storage key based authentication and authorization for data plan operations and regular old Azure RBAC for control plane operations. On the control plane side, you want to make sure to follow the principal of least privilege. Standard RBAC methods apply. Create groups and assign those groups a role and a scope that makes sense. For data plane operations, it's important to understand the authentication types and how each one gains authorization to interact with storage. There are two main authentication sources, storage account keys and Entra ID. There's also Active Directory as an option, but that's not an option for Terraform, so we'll skip that one. Storage account keys can interact directly with the storage account or generate shared access signature tokens. The storage account keys have implicit authorization to do anything on the storage account, so you should avoid using them if possible. If you do have to use them, you should rotate them regularly. Entra ID users and service principals can authenticate through Entra ID with Azure RBAC defining authorization at the

control plane and data plane through built-in or custom roles. This is a good option for service principals or users, and it's Microsoft's recommendation. You can add on Azure attribute-based access control to narrow down permissions to a specific path inside the storage container. One point of confusion is shared access signature tokens, or SAS tokens. Those can be signed by the storage account key or by an Entra ID entity. The token itself is a form of both authentication and authorization. Authentication is verified by the signer of the token. Authorization is baked into the token properties, and it can be scoped to a specific container or blob with certain allowed operations and a time limit. Tokens are a good option if your automation system can generate them. In terms of authorization for state data, Terraform expects to have list permissions on the container where your blobs are and write access to the blob itself. If you're using Azure RBAC with Entra ID, you can create a custom role with ABAC or use the storage blob data contributor built-in role assigned at the container level. For SAS tokens, you will bake the permissions into the token itself when it's generated. Now, how do you actually configure Terraform to use Azure storage for state data? Let's check it out. As you probably already know, state backend configuration is defined in a backend block inside of a Terraform block. The backend block syntax is the backend keyword followed by the backend type, in this case AzureRM, and then the backend configuration inside the block. When it comes to the backend configuration, all the arguments have to have literal values. You cannot use variables or expressions in the backend configuration. That's because the backend configuration is evaluated during initialization before any variables or expressions are known. You can hard code the values as shown here, or use a combination of environment variables and settings that are passed on the command line using the backend config flag. For the AzureRM backend, you'll need to specify the storage account name, container name, and the key at a minimum. The rest is going to depend on which authentication scheme you decide to use. You can use storage account keys, service principals, a managed security identity, or SAS tokens. For a service principal, you can specify a client's secret, certificate, or use OIDC for authentication. OIDC is a really good option if the Terraform automation platform you're using supports it. This removes the need to have a statically-defined credential like a secret or certificate. For service principals and MSIs, you can specify use Azure ADAuth, which will use the Entra ID authentication against the storage account. Otherwise, Terraform will try to retrieve the storage account key using the service principal or MSI. As I mentioned earlier, Microsoft recommends disabling the storage account key, so you should use Entra ID authentication. Now, I'm not going to cover every possible iteration here. Instead, I highly recommend checking out the official docs for the Azure RM back end. Honestly, I check the docs all the time when I'm configuring a back end just to make sure I'm not missing anything. If you've got existing state data that you need to migrate, the process is actually very simple. Once you've updated the backend configuration to use Azure Storage, simply run terraform init and follow the prompts. You'll be asked if you

want to migrate your existing state data to the new back end, and once you enter yes, the data will be moved to the Azure storage account. Migrating state data to a new back end is actually really easy, and to illustrate how easy it is, up next is a lab where you get to create an Azure storage account, configure an existing configuration to use the AzureRM back end, and migrate your state data to Azure Storage.

Importing Existing Resources

It's pretty unlikely that you are starting from scratch with Terraform. You probably have existing resources in Azure that you want to start managing with Terraform, so how do you bring those existing resources under Terraform management? Terraform has two workflows for importing resources, the declarative import process and the imperative import process. The declarative import process was introduced in Terraform 1.5 with import blocks and it's the preferred process for importing resources. The imperative process predates the creation of import blocks and can be used if you're on Terraform 1.4 or older, I'll briefly cover the imperative import process, but we'll focus on the declarative import process. The imperative process uses the command, terraform import, followed by the resource address and its unique identifier. When you run the command, it adds an entry to state data directly at the resource address specified, but it doesn't add any code to your configuration, matching that resource, you have to write that code yourself and then run a Terraform plan to verify that the configuration matches the imported resource. The imperative process is limited to a single resource at a time and makes changes to state without creating an execution plan first. That's an anti pattern for Terraform, so you should avoid using the imperative process, if possible. The declarative process uses the import block for importing existing resources. The syntax is the import keyword followed by a block containing the to argument with the address of the resource in the configuration and the id argument set to the unique identifier of the resource. Starting with Terraform 1.6, you can use expressions and non literal values for the arguments in the import block, and with Terraform 1.7, you can also use the for each meta argument to import multiple resources with a single import block. The import process uses the normal Terraform workflow of terraform plan and apply. The plan command includes an optional flag to generate the configuration for the imported resources automatically. This is an experimental feature as of recording, so you may receive some errors about the generated configuration when using it, but it does make the process of writing configuration blocks a lot easier than starting from scratch. Once the execution plan looks good, you can run terraform apply and the resources will be imported into state. After that, you can remove the import blocks from the configuration. The declarative import process allows you to import multiple resources at the same time, create an execution plan before altering state, and generate configuration blocks for you. It's definitely a step up from the

imperative process, but one thing it doesn't do is discover your existing resources and their unique identifiers. That's where the Azure Export for Terraform tool comes in. The Azure Export for Terraform tool is a open source project developed by Microsoft to help you discover existing resources in Azure and bring them under Terraform management, and I should point out that this tool is only for Azure resources. It does not extend to Entra ID, Azure DevOps, or any other resource outside of the AzureRM and AzAPI providers. When executing this tool, you can point it at an entire subscription, a resource group, or a single resource. It also supports filters for resource types, tags, and other patterns using the Azure Resource Graph query language. Once you run the tool, you can select which resources to import and it will generate a configuration for you, including import blocks, and then perform the import process to update your state data. There are a ton of options for the tool, so I highly recommend checking out the official docs. There's very little risk in using the tool against existing resources as it doesn't alter resources during the process. It just brings them under management by Terraform. If you're unhappy with the results, you can delete the config and state data and start over. You'll get to take Azure export for Terraform for a spin in the lab, so first, let's take a look at the declarative process of importing existing resources into Terraform.

Demo: Importing Resources with Import Blocks

In our role as a DevOps engineer at Globomantics, we've been tasked with managing the Azure infrastructure using Terraform, but the use of Azure predates our arrival. So we need to import an existing virtual network into Terraform. Let's see how we can do that using the declarative import process. We'll start by looking at the deployed Azure Virtual Network and subnets and documenting their unique identifiers. Then we'll create a Terraform configuration using the import blocks for the virtual network and subnets. Rather than writing the resource blocks ourselves, we'll use the generate-config-out flag to have Terraform try to do that for us. It probably won't get everything right, so we'll do a little clean up and then run a Terraform plan to verify that the configuration matches the imported resources. Once our plan looks good, we'll run terraform apply to import the resources into state. After that, we can remove the import blocks from the configuration. Let's head over to the demo environment and get started. I've deployed an Azure virtual network named hub-eastus using the Azure portal. It has two subnets in it named main and security. We'll need the unique identifiers for the virtual network and these two subnets. To get those, we can use the portal or the Azure CLI, but since subnets don't actually show their resource ID in the portal, we'll use the CLI instead. Now you might be wondering how do I know which property to use for this resource? 99% of the time it's the resource ID, but to confirm, we can look at the AzureRM Virtual Network resource in the AzureRM provider documentation. Down at the bottom, it has an Import section and the Import section will tell you which unique identifier you should

use, and sure enough, it's the resource ID. Now let's head over to the terminal and get the resource ID for the virtual network and two subnets. Over in VS Code, I'll bring up my terminal and I'll run the command az network vnet list g for the resource group it's in, which is hub networking, and then I'm going to add a query statement here to get just the resource ID, and then finally, o tsv, which sets the output to plain text instead of JSON. That will produce the resource ID of the virtual network, and now let's run a separate command to get the subnet IDs. I'll run az network vnet subnet list g with the resource group, vnet name, which is hub eastus, and then add the same query to get the resource ID and set the output type to plain text. That will give me the resource IDs for the two subnets. Now it's time to use these values in import blocks. I'll open up my configuration, it's in the import vnet directory over on the left and open up the main.tf file. I have the AzureRM provider block in there, and I'm going to add an import block for each of the three resources, starting with the virtual network. I'll create the import block and then set the ID to the resource ID and then set the two arguments to the address of the resource block where I want the configuration to be defined, in this case, azurerm_virtual_network.main. Now for the subnets, I'll create another import block, set the ID to the resource ID for the main subnet, and for the two arguments, I'll set that to azurermsubnet.main, and then I'm going to repeat the same process for the security subnet, copying the resource ID from down below and setting that one to azurermsubnet.security. With those import blocks in place, I'll maximize my terminal window, clear it, and run a terraform init to initialize the configuration and download the provider plugins. Once that's complete, I'll clear my terminal again and now run terraform plan with the generate config out flag. The flag takes an argument for the path to write the generated configuration to, so I'll set that to generated.tf. Once I hit Enter, Terraform will try and find the resources for each import block and generate a corresponding resource block using the to address that's in the block. And like I said, this is an experimental feature, and unsurprisingly, boom, we've got some errors. I'll open up the generated.tf file and take a look at the configuration. Scrolling through the virtual network block, our subnets are being defined in the virtual network, as well as in their own blocks, so I'll remove the subnets from the virtual network block. The flow_timeout_in_minutes argument needs to be in a range of between 4 and 30, but we actually don't need to define that argument at all, so I'll remove it entirely. In the subnets, the argument service_endpoint_policy IDs should be removed as well. Terraform does its best to populate the arguments based on what it finds in state data, but you'll probably run into issues like this if you use the generate config out flag. Also, all the values are going to be hardcoded, so you would want to add in some input variables and resource references. For instance, I'll change the VNet value in the subnet block to reference the virtual network block. Now, I'll bring the terminal back up and run terraform validate to make sure we've cleared up all the issues from the generated configuration, and hey, look at that, it comes back valid. Now I can run a terraform apply and check out the execution plan. I'll expand the terminal window up, clear it out, and run

terraform apply. Once the apply creates the execution plan, it looks like we have three resources to import, nothing to add, change, or destroy. That's exactly what we want, so I'll enter Yes. That is what we want to see. The goal is to import the resources without altering them. I'll approve the plan, and after a few seconds, our network and subnets are now being managed by Terraform.

Creating Networking Resources

Now it's time to lay down a foundation for future deployments with an application virtual network. We'll look at the network resources in Azure you'd commonly configure with Terraform, where you should place those resources, some popular modules for Azure networking that do some of the heavy lifting for you, and how to peer our application virtual network with a shared services VNet. You should already be familiar with Azure networking, but let's do a quick overview of the primary resources you'll be configuring. Everything starts with the virtual network, or VNet, that houses our other networking components. Inside the VNet, you will have one or more subnets that are used to segment resources. You can also delegate subnets to provide services like Azure Firewall or Azure Bastion or service endpoints to allow resources to communicate with other Azure services without going over the public internet. Traffic between the subnets and to other virtual networks or the internet is controlled by one or more route tables. Azure provides default system routes for common scenarios, and you can override or augment those routes by creating user-defined routes in route tables associated with your subnets. You can use route tables to direct traffic through virtual appliances or send traffic to a virtual network gateway for VPN or ExpressRoute connectivity. Traffic that is bound for the internet will require that virtual machines have a public IP address allocated or the presence of a NAT Gateway, similar to what you would have in an on-premises environment. To secure and control the flow of traffic, you can make use of network security groups at the network interface and subnet levels. NSGs allow you to control the inbound traffic based on source and destination IP address, port, and protocol. At the network edge layer, you can use Azure Firewall and Application Gateway to provide additional security and control. Azure virtual networks have a built-in DNS service that can be used to resolve names for resources in the virtual network. You can also create a private DNS zone for name resolution, especially useful for private endpoints or configure the VNet to use your own custom DNS servers. This is by no means an exhaustive list of networking resources, but it covers most of the common ones you'll be configuring with Terraform. If you want to know more, I suggest checking out some of the courses on Pluralsight that cover Azure networking. One of the challenges you'll encounter when developing your Terraform configurations is deciding where to place your network resources. Do you include the code for a virtual network in your application deployment? If you choose to separate out the virtual network and subnets into their own configuration, what about the network security groups, Azure Firewall,

or Application Gateway? Do you include those in the application configuration or in a separate configuration? There is no one right answer and a lot of it will depend on a few factors. Is the network resource specific to the application or is it shared? If it's specific, include it in the application configuration? If it's shared, put it in a separate configuration. Is the lifecycle of the resource tied to the rest of the configuration. If so, it probably belongs in that config. Otherwise, it should be placed in a configuration that has a lifecycle that matches the resource. For instance, the Azure Firewall is a shared resource that is used by multiple applications, so it shouldn't go in the AppConfig, but you may place an application specific rule collection in the application configuration. Is there a separate team in charge of the network resources? If you have a dedicated DNS team, they probably want to manage the DNS configurations. If you have a dedicated network security team, they probably want to manage the NSGs and firewall configuration. And lastly, who has permissions to make changes? A separate, but related question to the teams is who actually has permissions to make changes to the resources. It doesn't make much sense to include an application rule collection in your config if your team doesn't have permissions to make changes to the firewall. Evaluating the design and answering these questions ahead of time can save you a lot of headaches down the road in the form of refactoring and rework. Once you've decided which resources belong in your configuration, you can leverage a module to do some of the heavy lifting for you. Given the number of possible resources that go into a virtual network, it's no surprise that there are a lot of networking modules on the public registry. Let's take a look at a few of the most popular ones. From the main page of the Terraform Registry, I'll go into Browse Modules and then select the Azure checkbox to only view Azure related modules. Microsoft has developed two different networking modules that are called vnet and network respectively. These are very simple modules that will create a virtual network and subnets with the attendant settings like service endpoints and subnet delegations. The vnet module will also associate network security groups and route tables to the subnets if you provide the IDs. A slightly more robust module is the vnet module from squareops. In addition to duplicating the functionality of the vnet module from Microsoft, it also can create specialized subnets and route tables to handle private, public, and database subnet types. It will also configure monitoring and logging and set up a VPN gateway. If you plan to define routes and NSGs in the same configuration, this module could be a good option. I feel I'd be remiss if I didn't mention the Azure CAF module as well. This is a super module to be used in conjunction with Azure landing zones to provision everything in a subscription, and I do mean everything, including all of the networking. While this module is a bit of overkill for our purposes, if you're starting from scratch and you want to use landing zones, it's worth checking out. Another important consideration of managing your virtual networks is peering. There are two models for peering, hub and spoke and full mesh. In a hub and spoke model, you have a central virtual network that acts as a hub and one or more VNets that act as spokes. Spokes can communicate with each other through the hub, but not

directly. This is a good model for isolating resources and controlling traffic flow. In a full mesh, all the VNets are able to interact with each other directly, so you need to create a peering connection between each virtual network in the mesh and all other VNets. For small deployments, this might be viable, but I'm sure you can see how the number of connections rises quickly, not to mention to create a peering connection, you need to set up VNet peering on both participating virtual networks, effectively doubling the number of resources required to set up peering connections. Deciding which configuration will own the peering relationship is another point of decision. In a hub and spoke arrangement, you could have the hub VNet manage all the peering connections, but that will require updating the hub configuration each time a new peered VNet is being connected. Alternatively, you can include the peering resources in the spoke VNet configurations, that requires less configuration updates to the hub network, but it also requires that the account creating the spoke VNet has permissions to create a peering endpoint on the hub virtual network. Let's see the whole process in action with the demonstration.

Demo: Using the Network Module

In the previous demonstration, we imported an existing virtual network into Terraform, that was a hub VNet meant to provide shared services to application specific VNets through peering. Now, the folks at Globomantics are asking us to deploy an application VNet for their new app codenamed Taco Wagon. First, we'll leverage the VNet module in the Terraform public registry to create the application VNet and subnets. Then we'll create a NAT Gateway and routes to handle the internet bound traffic for the virtual network. Finally, we'll create a peering relationship between the application VNet and the hub VNet and verify that traffic can flow between the two subnets. Let's head over to the demo environment and get started. First, I'm going to start with a basic configuration that's defined in the app vnet directory. In the variables.tf file are basic input variables for things like a naming prefix, location, common tags, the virtual network address space, and subnet layout. The subnet layout is a map of strings where the name of the subnet is the key and the value is the address space. There is also a terraform.tfvars file which has values for input variables like the prefix vnet_address_space and subnet layout. In the main.tf file, I have an AzureRM provider block and a local block to set up the resource group and VNet names using the name prefix input variable. Below that is a block to create a resource group to house our new virtual network. I'm going to add a module block for the VNet module. Switching over to the browser, I'll grab the basic module block from the documentation and paste it right into our configuration. For the module name, I'll use app_vnet. Looking back at the documentation for the VNet module, it needs a resource_group_name, the location for the virtual network, and whether we want to use for_each for the subnets. Previous versions of the module used the count meta argument, which made it harder to reference

specific subnet resources. Heading back to the configuration, I'll set the resource_group_name to the name attribute of the resource group and the location to the resource_group location, and then I'll set the use_for_each argument to true. Going back to the documentation, below that it looks like we need to set the virtual network name and address space, so I'll add an argument back in our configuration for each one. The vnet_name will be set to local.value_vnet_name and the address space will be set to the input variable vnet_address_space. For our subnets, we need to configure the subnet_name and address prefix. To get the names out of our input variable, we'll use the keys function which returns a list of the keys in the subnet configuration variable, and for the prefixes, I'll use the values function instead. The last thing to add is the common tags using the tags argument. With that configuration in place, I'll pull up the terminal and run a terraform init to initialize the configuration and download this VNet module from the public registry. Once that's done, I'll kick off a terraform apply and see what Terraform wants to do. Once the plan completes, scrolling up, it looks like it's going to add four new resources that lines up with what I would expect, so I'll enter yes to accept the plan. Terraform will create the resources, and now the next thing we need to do is configure a NAT Gateway and associate that NAT Gateway with subnets to create the routes to handle internet bound traffic. Hiding the terminal again, let's get that NAT Gateway configured. First things first, I'm going to create a new file to hold our NAT Gateway resources called app_gw.tf. What goes in the NAT Gateway configuration? Well, we need a public IP address for the gateway to use, the gateway itself, an association between the public IP and gateway, and an association between the gateway and each subnet. I'm going to jump ahead a little bit to the completed configuration. The public IP address allocation needs to be static and the SKU needs to be standard. For the name, we can simply use the virtual network name and get the location and resource group name from the resource group itself. The NAT Gateway only needs a name, location, resource group name, and SKU. We'll go with standard for the SKU. The next block simply associates the public IP address with the NAT Gateway. The last block associates the NAT Gateway with each subnet, creating a route for the subnets to use the NAT Gateway. We'll use the for each meta argument and pass it the map of subnet names to IDs in the VNet module. Then we'll use each .value to get each subnet ID for the subnet ID argument. With that complete, I'll pull up the terminal again, run a format and validate to make sure I didn't enter anything incorrectly, and it looks good. Now I'll run a terraform apply with the auto approved flag, don't do that in production kids, and after a few minutes, the NAT Gateway is deployed. The last piece is to peer our application virtual network with the hub VNet from before. To do that, we need to create two AzureRM virtual network peering connections, one for the application VNet to the hub VNet and one for the hub VNet to the application VNet. We will need the hub VNet's name, resource group, and ID. Fortunately, we can look up the ID with a data source of we have the other values. Over in input variables, I have added the variable hub_vnet as an object with two attributes, name and resource_group_name. I'll

create a new file called peering.tf to hold the peering resources, and we'll fast forward a little bit to when the configuration is complete and review the contents. First, we'll have a data source for the hub virtual network. The name will be set to our new input variable hub_vnet with the attribute name, and our resource group name will be set to the input variable hub_vnet.resource_group_name. Now we can reference that virtual network using our data source. For the first peering resource, we'll have spoke_2_hub. For the name, we'll give it the name of the application vnet2 and then the name of the hub VNet we're connecting to. The resource group name will be equal to the resource group of our application VNet, the virtual network name will be the name of our application virtual network, and the remote virtual network ID will be the ID of the hub network which we can retrieve through the data source. The second peering connection goes from the hub network back to the spoke, and so this one, the name will be the hub network2 and then the name of the application VNet. Below that, the resource_group_name will be the resource group that holds the hub_vnet, and the virtual network name will be the name of the hub virtual_network. The remote_virtual_network_id will be the resource ID of the application virtual network. With our configuration updated, I now need to update our terraform.tfvars to include a value for the hub_vnet input variable, so I'll add that entry. The name should be equal to hubeastus, that's the name of our hub VNet, and the resource_group_name, if you remember, is hubnetworking. With all of that in place, I'll pull up the terminal and run a terraform validate to make sure that the configuration is valid. Alright, it came back green. Now let's run a terraform apply to create the peering connection The apply has completed, and if we head over to the browser and look at our hub virtual network, clicking on peerings, there's the peering connection to the application virtual network and it says connected. Good stuff. Now it's time for you to try it yourself. Head on over to the lab environment and create an application virtual network and peer it with a shared services virtual network.

Deploying an Application (Azure Compute)

Deploying an Application

Building out a network in itself is not very useful. You're probably going to want to deploy an application or service on top of that network. In this section, we'll be looking at how to deploy an application on Azure using Terraform. We'll start by looking at the different types of Azure virtual machines and the resources that are available to us for virtual machine creation. It's more than just the VM itself, it's also network interfaces, disks, and other resources that we'll need to consider. Because there are several resources tied up with the Azure Virtual Machine, we're going to take a look at the Azure compute module that puts together those components to simplify the deployment process. Next, we'll look at how we can

deploy Virtual Machine Scale Sets along with a load balancer. The Load Balancer, in particular, is a tricky beast, so we'll employ a module to help us out. First, let's start off with how to create a virtual machine in Azure using Terraform.

Creating an Azure VM

While there are many compute options in Azure, including functions, container groups, AKS, and more, most folks start with the humble virtual machine and that's where we'll start too. There are essentially two options for creating virtual machines in Azure. You can create a single virtual machine or a Virtual Machine Scale Set. The single VM needs some other components to be functional, in particular, a network interface. The Virtual Machine Scale Set, on the other hand, includes a network interface as part of the resource. First, we'll take a closer look at the Azure VM resources, and later, we'll cover VM Scale Sets. When creating an Azure VM with Terraform, there are actually three different resource types available. There's azurerm_virtual_machine, azurerm_linux_virtual_machine, and azurerm_windows_virtual_machine. The first type azurerm_virtual_machine predates the other two types and has been superseded by them. The azurerm_virtual_machine resource type is still available, and you'll probably see it in older configurations, but it is feature frozen, meaning it isn't going to get any of the new features that are added to the other two resource types. There are two reasons to still use the older resource type. First. If you need to attach unmanaged disks to your virtual machine, the newer resource types don't support that. Second, and related to unmanaged disks, if you want to attach an existing OS disk to your virtual machine, you'll need to use this older resource type. The newer resource types only support captured images. If neither of those situations apply, I'd recommend using the new resource types so you can get full access to all the new features that are being added to Azure VMs. The features of Linux and virtual machines are different enough on Azure that the provider maintainers decided there should be two separate resources. While each resource has many of the same arguments, they differ in some key areas. For starters, the Windows VM requires an admin password, whereas the Linux VM does not, provided you supply an admin SSH key instead. The Windows VM does not have an admin SSH key argument. The Windows VM has a bunch of arguments that are specific to Windows functionality such as WinRM listener, enable automatic updates, and hot patching enabled. Windows also has an additional unattend content argument that allows you to add additional content to the unattend.xml file that is used to configure the VM during deployment. We'll talk about customizing the VM deployment later in the lesson. It does seem like the main reason to have two different resources is the Windows features, although that may change in the future. Definitely check out the documentation for the most up to date information. Beyond those differences, the resources are essentially the same. You need to specify a size, a location, and resource group for the virtual

machine. You'll need to specify the source image to use for an operating system and information about the operating system disk, and finally, one or more network interfaces to be attached to the virtual machine. Beyond the Azure Virtual Machine, there are other related resources you'll need to know about. The first is the network interface, azurerm_network_interface. This resource is used to define a network interface and associate it with a subnet and possibly a public IP address. To associate a network interface with a virtual machine, you specify in the network_interface_id's argument of the VM resource. You may wish to attach one or more data disks to the virtual machine. Those are created using the azurerm_managed_disk resource and then attached to the Azure VM using the azure_virtual_machine_data_disk_attachment resource. Boy, that's a mouthful. This is in contrast to attaching a network interface which is done directly in the VM resource. The older Azure virtual machine resource allowed you to specify data disks directly in the VM resource, but that's not the case for the newer two resource types. Diagnostics are pretty important for when things inevitably go wrong with your VM, and the boot diagnostics argument can either use an existing storage account or create a managed one for you to use. If you plan to use your own storage account, you'll need to create one or have one ready. The last thing I want to mention is an identity for the VM. You can associate a system managed or user managed identity with the virtual machine. The identity can be granted permissions through Azure RBAC to interact with other services in Azure. If you plan to leverage a user managed identity, you'll have to create an identity using the azurerm_user_assigned_identity resource and then associate it with an identity block inside of the virtual machine resource. That's not every related resource, but it's some of the most important to be aware of when creating an Azure virtual machine. Virtual Machines and VM Scale Sets aren't very useful with a vanilla OS install. You're probably going to want to install an application of some kind and configure the VM to run that application. There are several ways to do this, and I'm going to try and cover the most common ones in fairly rapid succession. When possible, you should use a custom image that has your applications and requirements preinstalled. This will make the spin up time for your VMs much faster and help avoid configuration drift. You can create a custom image using HashiCorp's Packer or leverage the Azure VM Image Builder service. If a custom image isn't an option or you need to do some additional configuration after the VM is deployed, you can use the custom data or user data arguments to pass scripts to the virtual machine. The user data argument is a newer version of the custom data argument that persists for the lifetime of the virtual machine and can be updated after the VM is created. The custom data argument is only used during the initial deployment of the virtual machine. Generally, user data is the better choice, but custom data is still available if you need it. Azure also provides the ability to run scripts through their Azure custom script extension. You can use the azurerm_virtual_machine_extension resource to define a custom script to run on the virtual machine. You can also use other configuration management tools like Ansible, Chef, or Puppet to configure your virtual machine. Some of these tools

require an agent to be installed on the VM, while others require SSH or WinRM to be enabled. The main takeaway here is that there are many ways to perform configuration of your VMs after deployment. Terraform is not a configuration management tool, so I'd recommend using a dedicated tool if your configuration needs are complex. Now that you have an idea of the various options involved in provisioning Azure VMs, let's simplify the process by using the Azure compute module.

Demo: Moving to the Compute Module

The virtual networks we've provisioned in previous sections are pretty lonely. They need some applications running to make them feel whole. In this demo, we'll deploy an Azure Virtual Machine with the virtual machine module. Globomantics would like us to start by deploying a simple virtual machine, and they'd like us to leverage the virtual machine module for deployment. We'll take a look at that module and a newer module from Microsoft that's not quite ready for production use. Then we'll copy an example of usage for that module into a local configuration and update the configuration with the necessary input variables and outputs. Globomantics has an example web application they'd like us to deploy, so finally, we'll deploy an Azure VM to the application virtual network using the module and feed it that basic web application via the user data argument. Let's head on over to the Terraform public registry to get started. In the Terraform public registry, I'll go into Browse Modules and then browse modules selecting just Azure type modules, and we can search on compute to get all of the compute modules related to Azure. Just like the networking modules, there are several available modules for spinning up an Azure VM. The one we'll focus on is the virtual machine module developed by Microsoft. There's also an ongoing project called Azure Verified Modules for Terraform which also has a virtual machine module, but that is only on version 0.6.0, so it's not quite ready for production use. Azure verified modules for Terraform is an exciting new project from Microsoft to provide consistent and supported modules for Azure infrastructure, so while we'll use the older virtual machine module for this demonstration, I'd recommend keeping an eye on the verified modules as they near 1.0. Heading back to the virtual machine module, there are 63 possible inputs for the module. That's a lot of options. What if we simply want to deploy an Azure VM running the latest version of Ubuntu. When working with Azure modules, it's often easiest to look at some of the examples to get an idea of what's expected. To that end, let's go look at the source code for this module. Over on the GitHub repository, we'll look inside the examples directory and in there is a basic example. That sounds pretty promising. Our basic example seems to include network security groups and disk encryption, but I suspect the actual module block is in the main.tf file. Inside the main.tf, there is the optional creation of a resource group followed by the use of the vnet module to create a virtual network. Hey, that looks pretty familiar, doesn't it? Below that is the creation of

SSH keys for use with the Linux VM and a public IP address to use as well. Then we have the actual module usage itself which is creating a Linux VM. I'll note the important pieces that apply to our deployment. We need to set the image_os to Linux and set the location and resource group name. We won't have any data disks, so we can skip that part. We do want boot diagnostics, but without the customer managed key, so we'll need to take a closer look at that argument. We also want a network interface, and we can pass it to the public IP address as shown here. Below that is the admin username and SSH keys and then the name and OS disk properties. The last part takes the place of specifying image information. Os_simple lets us use the value, UbuntuServer to get what we're after, the latest version of Ubuntu Server, that with the size and subnet_id, finishes up the configuration. I'm going to copy the contents of this main.tf file and move them into a new configuration I have in VS Code. Over in Visual Studio Code, after a little bit of coding magic, I have the copied file customized in VS Code along with variables and outputs. Looking inside the directory app vm and inside the main.tf, I added a provider block and made the creation of the resource group part of this configuration. We're still creating a public IP address, although it has to be standard and static to be in the app subnet for our application virtual network, and that's because we have a NAT Gateway association with that subnet. Next, we have the actual module block and a lot of it, I kept the same only tweaking certain values. Instead of using the new storage account argument for diagnostics, I switched to boot diagnostics set to true. This will create a managed storage account for diagnostic logging. I also added an input variable for the admin username and the virtual machine name, and since the subnet_id will be coming from our application virtual network, I added an input variable for that value as well. Finally, to configure the virtual machine, I'm using the user data argument to have it execute the contents of a custom_data.tpl file in the templates directory. The text for user data must be Base 64 encoded, which is why that function is being invoked once the template file is rendered. The script will start a basic Flask app on the port we specify with the application port variable, so we need to open that port for traffic using a network security group and NSG rule, that's what the next two blocks do, and then we need to associate the NSG with the network interface of the machine. This is a perfect example of when you would want to include your NSGs with your virtual machines. The port is being defined with an input variable and both resources can access that port value to keep things consistent. At the bottom of the main.tf is a block to create a file containing the private key locally. We don't need to SSH in because we're using user data to configure the virtual machine, but if you want to access it directly, you can do so by adding an NSG rule that allows port 22 and logging in with the private key. For the outputs, we are exposing the public IP address so we can browse through the Flask app and make sure it's working. Let's get this thing deployed. First, we need to define values for the input variables that don't have a default value. I've got an example, terraform.tfvars file with placeholders in it. I'll make a copy of that file and name it terraform.tfvars, and we need to give a resource_group_name, vm_name, and subnet_id. The resource_group_name can be

tacowagon app vm, and the vm_name can be tacowagon app. For the subnet_id, we can grab that from the previous demo. I've added an output to that configuration to emit the subnet names and IDs. By running Terraform output, I can grab the WEB_SUBNET_ID and paste that into the terraform.tfvars file. When the value is staged, I'll run a terraform init to pull down the provider plugins and the virtual machine module. Once that's complete, I'll run a terraform validate just to make sure I didn't miss anything. That looks all green, so now we can apply the configuration. The deployment will take a while since it needs to spin up the virtual machine, and even after that completes, it will take a minute or two for the Flask application to finish installing. Jumping ahead to the completed deployment, I'll copy the public IP address from the outputs and head on over to my browser. I'll browse to HTTP and then the public IP address, and after a few moments, the page loads. Sweet. We've now deployed an Azure VM using the virtual machine module, but that's just one virtual machine. What about a scalable web front end? That's where VM Scale Sets come in, which is what we'll cover after the lab.

Using an Azure VM Scale Set

Virtual Machine Scale Sets are a way to define a set of nearly identical virtual machines that are managed as a single resource. The number of VM instances can be adjusted manually or automatically based on demand or a schedule, Just like the Azure VM, there is more than one resource type for VM Scale Sets. Also like the Azure VM, there is a deprecated Azure_virtual_machine_scale_set resource that is feature frozen and shouldn't be used for new configurations. The only reason to use it is if you need to attach unmanaged disks to your VMs. Beyond the deprecated azurerm_virtual_machine_scale_set resource, there are three other resource types
available, Azurerm_linux_virtual_machine_scale_set, Azurerm_windows_virtual_machine_scale_set and azurerm_orchestrated_virtual_machine_scale_set. The first two support Linux or Windows VMs respectively, and the same differences we saw in the single VM resources apply here. The
azurerm_orchestrated_virtual_machine_scale_set resource was developed
to support a different orchestration mode for VM Scale Sets. It's a complex topic that we won't cover in detail for this course, but here's the gist of it. There are two orchestration modes in Azure VM Scale Sets, uniform and flexible. As implied by the name uniform mode means that all the VMs in the scale set are essentially identical. This is the original mode for VM Scale Sets, and it uses the VM Scale Set API, instead of the more general Azure IAS VM API. The API comes with certain limitations in terms of functionality and features. The
azurerm_linux_virtual_machine_scale_set and
the azurerm_windows_virtual_machine_scale_set resources are for VM Scale Sets in the uniform mode. Uniform mode is great for nearly identical workloads
that don't need the extra scale and features of flexible mode. The flexible mode

for VM Scale Sets uses the full Azure IAS VM API and allows you to mix and match different types of VMs in the scale set. You can mix different sizes, spot, and regular instances and even Linux and Windows VMs in the same scale set. You also get full access to Azure VM features like Azure Backup and Azure Site Recovery, and flexible mode supports higher scale limits than uniform mode. The azurerm_orchestrated_virtual_machine_scale_set resource is for VM Scale Sets in flexible mode. Which one you should choose is beyond the scope of this course, but now you know why there are four different resources in the provider and how they differ. Of course, if you're going to have a scale set of virtual machines, you will probably want to load balance them in some way, so let's dig into Azure load balancing next.

Adding an Azure Load Balancer

The word load balancer can be a bit overloaded. Basically, it's the process of distributing requests across one or more nodes of a system. Azure has four basic options when it comes to load balancing, Front Door, Traffic Manager, Application Gateway, and Load Balancer. I'll admit calling one of the options load balancer does make things a bit more confusing. Front Door and Traffic Manager are both global services meant to balance traffic across regions and provide the best response time for requests. Front Door can handle HTTP traffic only, while Traffic Manager can handle any kind of traffic as it is a simple DNS request response mechanism. Application Gateway is Azure's Layer 7 load balancer and it operates at a regional level. It can perform advanced functions like SSL offload, path based routing, and provide a web application firewall. Load Balancer is a Layer 4 load balancer, and it can operate at a regional or cross regional level. The cross region version of the Load Balancer is assigned a unicast IP address which it routes to load bouncers created at the regional level. Regional load balancers can accept any type of web traffic, but they lack the more advanced features of the Application Gateway. For our purposes, we will be working with the load balancer, but bear in mind the other options, especially if you're scaling your web application to thousands of nodes or need a robust front end to do filtering, routing, and offload. The load balancer is not a single monolithic resource in Terraform. Instead, it is expressed as several components that work together to form a working load balancer construct. The front end of the load balancer accepts incoming traffic and has the front end IP configurations. This could be a public IP address for public facing applications or a private IP address for internal load balancing. A single load balancer resource can have multiple front ends. On the back end of the load balancer are one or more back end pools. This will be the machines running your applications, and they can be added via network interface or IP address. The thing that ties the front end and back end together is a load balancer rule. It defines the front end port and protocol to listen for traffic on and the ports and protocols on the back end pool to send traffic to. Also part of the rule will be a load balancing

algorithm to determine how traffic is distributed amongst the healthy backend nodes. How does it know which nodes are healthy? Through a health probe. A health probe defines the port and protocol to test what request path should be used, if applicable, and logistics on how to decide when a probe is successful or failed. That decision includes how often to send a probe to the backend endpoint, how many failed attempts should remove an endpoint from rotation, and how many successful probes should allow it back in. Setting a higher probe threshold can prevent endpoints from being added that are in an unstable state. Load balancers can also support NAT inbound rules where traffic that hits the frontend IP address on a specific port is always directed to the same backend virtual machine, and the load balancer can also support outbound NAT so that machines associated with the backend pool can use the load balancer for NAT based access to the internet. Those are the major components of the load balancer, but there's one other very important component we didn't mention, and that's network security groups. The network cards associated with VMs have a deny by default rule. What that means is if there is not a network security group rule that allows traffic for a given port and protocol, the virtual machines won't accept it. I can't tell you how many times I have set up a backend pool and all my health probes were failing because there was no NSG rule allowing traffic from the load balancer to the members of the pool. In order to get traffic flowing, you'll need one, possibly two rules. The first is to allow traffic from your source to talk to the VM Scale Set on the correct port and protocol. The load balancer will pass traffic through with the original source IP address, so you need to allow traffic from that source IP address to reach the port and protocol of the application. If the first rule doesn't allow traffic from all source IP addresses, AKA the internet, then you'll need to add a rule allowing traffic from the load balancer to reach the correct port and protocol on your VMs. With that in mind, why don't we walk through an example deployment in the demo environment, combining both the VM Scale Sets and an Azure Load Balancer.

Demo: Create an Azure VM Scale Set and Azure Load Balancer

We deployed a simple Azure VM in the previous example, but now Globomantics wants the ability to scale the application through the use of VM Scale Sets and a load balancer. We will start with a high level design of what we want to deploy. Then we'll review the configuration for the VM Scale Sets and load balancer. Finally, we'll deploy the configuration and verify that it works properly. Let's start with the design. We already have an application VNet with a subnet dedicated to web services. We will deploy the VM Scale Set into that subnet. The application deployment will be a flask application running on a non standard port, so we need to make that port number configurable. To provide access to the VM Scale Set, we are going to add a public facing Azure

Load Balancer. The front end will have a rule that listens for traffic on port 80 sends it to the custom port on the VM Scale Set backend pool. We also need to include a health probe to make sure the application is ready on each instance. The health probe can be used by both the load balancer and the VM Scale Set to determine instance health. And finally, can't forget about that NSG. We will allow traffic from anywhere to access the application on the custom port for the application, that will cover traffic from both the internet and health probe traffic from the load balancer. Now let's head over to the configuration and review it. Our virtual machine scale set is going to take a lot of cues from the previous Azure VM demo, but rather than using a module for deployment, we'll simply use the azurerm_linux_virtual_machine_scale_set resource. The configuration we'll be working with is in the app vmss directory. I'll expand that and open the main.tf file. Our VM Scale Set is going to take a lot of its cues from the previous Azure VM demo, but rather than using a module for deployment, we'll simply use the azurern_linux_virtual_machine_scale_set resource. Inside the block is an instances argument that defines how many VM instances the VM Scale Set should start off with. The source image will be the latest version of Ubuntu 22_04, just like the Azure VM from before. Unlike the azurerm_linux_virtual_machine resource, the network interface resource is defined inside a nested block for our VM Scale Set. This is where we can configure the network security group to attach to the NIC, and in the IP configuration, we can describe which subnet to attach to and which backend pool to associate with this IP address. Since we are going to create a health probe for the load balancer, we can also use that health probe to check on the health of the VM Scale Set instances. And then lastly, the user data is the same as the Azure VM taking a port and an admin username argument. That does it for the VM Scale Set. What was several resources, the VM, the NIC, and NSG to NIC association is all included inside the single VM Scale Set resource. Below the virtual machine scale set resource we have a module for deploying a load balancer. Remember that a load balancer is actually several components, so a module is a perfect construct for grouping those components together. For our load balancer, we are setting the type to public which will provision a public IP address resource for the front end. We will use a standard SKU for the public IP address with a static allocation method. The lb_port argument describes a load balancer rule to create on the load balancer. This one will send TCP traffic on port 80 to the custom application port we defined. The module creates an implicit backend pool as a target for the rule, and this is the backend pool that we associate in the IP configuration of our VM Scale Set instances. The lb_probe argument creates a health probe of type HTTP, checking the application port at the path slash. The module uses a count argument to create both the load balancer rule and the probe, so the first probe in the list will be associated with the first rule in the lb_port list. Since there's only one of each, we know the HTTP probe will be applied to the lb_port rule above it. Because the module uses a data source to access the resource group, we have to put a depends_on for the resource group to ensure it exists before the module is created. The last two entries create the

network security group and security rule that allows traffic from anywhere to the virtual machine scale set instances over TCP and the application port. That does it for the resources. The input variables are very similar to the Azure VM configuration. The main difference is that we're using a naming prefix with a validation rule and there's also a vmss_count variable to set the number of instances to create. The output file includes a single output value for the public IP address, so we can browse to the website once the instances are up. So let's get this sucker deployed. Many of the input variables have valid defaults, but there are two values we need to set from the terraform.tfvars.example. The subnet id is going to be set to the web subnet of our application VNet and the application port, I'm going to set to 8080 instead of the default of 80. Might as well try out a non-standard port, right? I'll create a copy of the terraform.tfvars.example file and rename it just terraform.tfvars, and then update the subnet_id with the subnet_id of the WEB_SUBNET from the application virtual network. Just as I did before, I'll go into the configuration for the app VNet and get the outputs to see the ID of the web subnet, and then I'll grab the WEB_SUBNET_ID and paste it into the terraform.tfvars. Now we're ready to run terraform init. I'll navigate into the proper directory and then run terraform init to download the modules and providers. Once that has completed, it's time to validate the configuration to make sure it's in good shape. That comes back green, so let's deploy the configuration now. The apply process will take a while as it doesn't consider the VM Scale Set deployment complete until the initial number of instances have been spun up successfully. Once it's done, it may still take a few moments for the HTTP health probe to go green. Let's jump over to the portal and monitor the status of the instances. Here is our virtual machine scale set in the portal. And I'm going to go into the Instances portion to see the status of the current instances. Azure will create more than the initial number of requested instances since some of them may fail. Once it has two that are up and functional, it'll flush away the rest that are not needed. After refreshing the page, it's narrowed it down to the two instances and gone forward with the installation and update process. One is currently running and the next one is in the process of updating. Once it's up and running, the health probe will start checking to see if the website is up and available. I'll refresh the page again to see the health state status. We're still waiting for the health state to move into a healthy status. This may take a few moments as the Flask app installs, so I'll jump ahead to when the health state has moved to healthy for both instances. Okay. After a few moments, both of our instances are up and running and reporting a healthy state. If we go back to the terminal in VS Code, Terraform shows that our deployment completed successfully. Let's grab that public IP address from the configuration output and paste it into our browser. And after a few moments, the site loads. Excellent! We have successfully deployed a VM Scale Set and load balancer in Azure with Terraform. Now it's time for you to give it a try. Give the next lab a go and try setting up your own VM Scale Set and load balancer. Good luck.

Automating Your Deployments (Azure DevOps)

Automating Your Deployments

In the previous sections, we've been deploying our infrastructure using the Terraform CLI. It's great to be able to deploy infrastructure from code, but as you mature in your use of IAC, you'll want to start automating your deployments. This is where a CI/CD pipeline comes in. In this section, we'll look at how to automate your Terraform deployments using Azure DevOps. We will start this section by reviewing the standard CLI workflow for Terraform and then look at the considerations for automating your Terraform deployments. Terraform has some specific environment variables you should set for automation, and you also need to determine how to handle remote state and input values. Then we'll take a look at how to build a GitOps pipeline for Terraform. There's more than one way to structure a repository and handle multiple environments, and we'll look at some common patterns for managing your Terraform code and triggering deployments on your pipeline. Once we have the source control concept sorted, we can look at how to define a pipeline in Azure DevOps. We'll look at the different options for defining a pipeline and how to connect your Azure DevOps to your source control and Azure environments. Our pipeline is going to need some input values for our Terraform code, and we'll look at some options on where to store these values, including Azure Key Vault and AppConfig. By the end of the section, you'll be ready to build out your own Azure DevOps pipeline for Terraform.

Creating an Automation Workflow for Terraform

Automating Terraform deployments is more than just scripting out the same commands you'd run at the command line. While the overall workflow is the same, you need to take into consideration that no one is there to answer any prompts or provide input, so before we automate, let's review the standard Terraform workflow. Your usual Terraform workflow starts with writing some new code or updating your existing code. Once you have your code ready, you run terraform init to download the providers and modules you need, then you would typically run terraform fmt to make sure everything looks pretty and terraform validate to check for any syntax errors. Then finally, terraform plan to see what changes Terraform is going to make. This process of adding code, validating the config, and running a plan is part of a development cycle that you'll probably repeat several times before you're ready to deploy or update your actual infrastructure. We are not trying to automate this portion of the process. This is simply what you'll be doing locally as you develop your code, but we do want to verify that all these steps have been followed before we deploy. Once you're ready to deploy for real, you'll run terraform plan and then save the plan file to verify that the changes are what you expect. You might have

someone else review the plan or simply check it yourself. Once the plan is approved, you can run terraform apply to make the changes. This is the portion of the process we're aiming to automate. When you're automating your Terraform deployments, you need to consider how to handle the steps that normally require human input. We'll deal with remote state and input values in a moment, but first, I want to focus on a few other settings you should consider when automating Terraform. Terraform has some built in environment variables that you can use to control how it behaves when running in an automated environment. These are TF_IN_AUTOMATION set to true will indicate to Terraform that it is running in an automated environment. This will suppress any helpful prompts Terraform sometimes displays, and it will make the logs easier to read and parse. TF_INPUT, when set to false, suppresses any prompts for input values. This is useful when you're running Terraform in an automated environment, and you don't want to be prompted for any input value since there's no one there to provide those inputs. TF_PLUGIN_CACHE_DIR sets the directory where Terraform will cache all the plugins it downloads. You can set this to a directory that's shared between your pipeline runs to speed up the plugin download process. TF_LOG is a log level setting that you can use to control the verbosity of the logs. You can set it to trace, debug, info, warn, or error. You probably want to set this to at least info or maybe even debug to make sure you have sufficient logs to troubleshoot any issues. TF_LOG_PATH is the companion to TF_LOG, and it's the path to a file where Terraform will write its logs. Depending on how your automation solution captures terminal output, you might want to set this to a file that you can review later. When you're automating your Terraform deployments, you'll need to locate your state data on a remote back end. Honestly, as we reviewed in a previous section, you should be using remote state anyway, so it's more a consideration of how you pass the remote back end configuration to Terraform during initialization. The most common approach is to have a partial configuration in the code that sets the backend type, but omits the rest of the configuration. During the initialization step, you can pass the rest of the configuration using the backend config flag. The actual backend settings still need to be stored somewhere though. Basically, all automation pipelines have a way to store configuration values. Azure DevOps pipelines has a built in library where you can store variable groups or secure files. You can also integrate with Azure Key Vault to pull secrets directly into your pipeline. Input values for variables are another consideration, but we'll dig into that later. Zooming back out to the big picture. Our automation workflow will look something like this. Step one, initialize Terraform with the back end configuration stored in our pipeline. Step two, run terraform fmt and terraform validate to check the code. Step three, run terraform plan and stash the plan file for review. Step four, approve the plan through some kind of manual or automated process, and then step five, run terraform apply to make the changes. You can add more steps to this process, but I would say that this is the minimum you should have in the pipeline, but where is the code coming from? How do we trigger the pipeline, and how do we handle different

environments? That's what we'll cover in the next lesson.

Building a GitOps Pipeline

Infrastructure as code is exactly that, it's code, and just like any other code, it should be stored in source control. A popular automation pattern is to use the events that occur in source control to trigger actions in CI/CD, AKA GitOps. This is not a course on Git, so we're not going to go in depth on Git operations or repository management. That being said, there are a few key concepts you need to be aware of if you're going to implement a GitOps pipeline. While Git is not the only form of source control, it is certainly the most popular, and it is built upon the idea of a versioned source control repository. You interact with your local repository by making changes in the working directory, then staging those changes with git add, and then finally committing the changes with git commit to the local repository, essentially creating a new version of your code identified by a commit hash. You can also create separate branches of your code to work on new features or ideas without impacting the main branch of code. There's a lot more to it than that, but I'm trying to keep things simple. Your local repository can have a corresponding remote repository that it pushes changes to and pulls changes from. The remote repository can be shared with others who all have their own local copy of the repository that they keep in sync with the remote one. Let's take a closer look at the branches using a typical code development workflow. In our example, we have our default branch, often called main with the most recent commit we'll call L0 which corresponds to the remote commit we'll call R0. When you want to update your code, you can create a new local branch and test out some changes. This avoids impacting the contents of the main branch. Then you can commit those changes, creating a new commit we'll call L1 and then push the new branch up to the remote repository where further testing might be kicked off automatically, then you create a request to merge the contents of your new branch into the default branch on the remote repository. You or someone else will review the request, some more automated tests might kick off, and then the request can be approved and your feature branch changes are merged into main, creating a new commit we can call R2. Assuming you're done with the remote feature branch, you can now delete it, and lastly, to close the circle, you would pull the updated main branch back down to your local repository, creating a local commit called L2 and then delete the local feature branch and then start the development cycle over again. There's a lot more to it than just that, and if you wanted to know more about Git, there's an excellent getting started course here on Pluralsight, but now you know the key workflows and concepts of branch, commit, push, merge, and pull. How does that fit into our standard Terraform workflow? The key portions of Terraform code development are updating the code, validating the code, previewing changes, and applying changes. Let's try to put that workflow into a GitOps context. On your local repository, you want to make an update to your Terraform configuration, so you should first create a new

branch, so the main branch is unaffected. Then you'll make your changes to the code and do some basic validation tests to make sure the code is syntactically and logically sound. If everything looks good, you can push the branch up to the remote repository. Now you're ready to preview the changes. You can do that through a pull request. You'll create a request to merge the changes into the main branch and have your pipeline take that as a signal to run Terraform plan against the environment and log the execution plan. You or someone else can review the plan, and if it's not what you want, you can make changes again locally and push those changes up to the remote repository. Your pipeline will see a new commit on the open pull request and kick off a fresh plan. The cycle repeats until everyone's happy with the results, and then the change is merged into main, and a terraform apply runs to make the proposed changes in the actual environment. Once the change is complete, you can pull the updated main branch back down to your local repository and use that to create the next branch for new changes, but this isn't the only workflow to implement GitOps with Terraform, but it's a pretty common base. You can expand on this base by adding automated testing, security scans, policy as code, and multiple environments. Let's try putting this theory into action by building out an Azure DevOps pipeline for Terraform.

Demo: Using Azure DevOps Pipelines

Globomantics uses Azure DevOps to run their application pipelines, and they don't see why Terraform should be any different. They would like us to check our Terraform code into Azure Repos and use pipelines to automate the process. In this demonstration, we'll start by creating a project in Azure DevOps and look at the initial settings. To successfully deploy our infrastructure to Azure, we'll need to connect pipelines to our Azure subscription. We'll walk through the steps of creating a service connection. We'll be using Azure Repos to store our code, so we'll get a repository created for our app virtual machine scale set configuration and push it up to the project repo, and then we'll move our state to remote storage and grant proper permissions for the service principal Azure DevOps will be using. Let's get started. Alright, in my browser, I'm logged into Azure DevOps and I'm going to create a new project. This project is going to house our repository using Azure Repos and our Terraform automation in pipelines. I'm already at the new project screen, so I'll call the project tacowagon WRITE TO US app WRITE TO US vmss. For the visibility, I'll leave it as private, and under the advanced options, I'll leave it using Git and Agile and then click on Create. Once the project has been created, it will take us directly to the project overview screen. By default, our Azure DevOps project includes boards, repos, pipelines, test plans, and artifacts. We're really only going to be using repos and pipelines, but it's fine to leave the other services enabled. Next, we're going to connect this project to our Azure subscription. From the project overview, I'll go into project settings and select service connections. This is how you can connect

your project to one or more Azure subscriptions as well as other services. I'll click on Create service connection and select the type Azure Resource Manager. On the next page, there are a ton of authentication options. If you have an existing identity that you want to use, you can just do that by selecting one of the manual options. My recommendation is to use the recently released workload identity federation option, whether it's automatic or manual. This uses OIDC for authentication, which as we covered in a previous section, creates short lived tokens and uses the trust between Azure DevOps and Entera ID as part of the authentication. After that page, it will want to load the subscriptions in Azure that I have access to. After a moment, I'll be presented with a login page which will use my credentials to enumerate the Azure subscriptions. I'll move that off screen and sign in with my account. Among the subscriptions, it has selected the correct one already, so I'll give the service connection a name like tacowagon app, and under Security, I'll grant access to all pipelines. That way, when I create a new pipeline, I won't need to explicitly grant the pipeline access to use the service connection. Depending on your security requirements, you may want to force the assignment to be explicit per pipeline. Once I click on Save, Azure DevOps will use my account to create the federated identity and grant it the contributor role on the subscription we selected for the service connection. If you want to tweak that, going into the service connection properties, there are links to configure the role assignments and manage the service principal. I don't need to adjust the role, but let's check out the service principal. Clicking on that link will take us to the overview page of the service principal. If you're curious about the federated credential, going into certificates and secrets shows us the federated credential, and in the properties of the credential are the Azure DevOps token issuer and a subject identifier linked to the project and service connection name. Any request that doesn't originate from the token service with this subject identifier would be denied. This prevents the need for passwords, certificates, or long lived credentials hanging out in your Azure DevOps project. I like that. Now, we need to set up our code for the repository, and we'll start that process in VS Code. In VS Code, we are going to create a new local repository using the code from our existing app vmms deployment. In VS Code, I'm going to launch the terminal and navigate to the app vmss directory. Then I'll run the command git init b main to initialize a new local git repository with a branch called main. Our new local repository is empty, so let's add our Terraform config. Before we do that, I want to note that I added a .gitignore file to the directory. This prevents Git from adding file patterns listed in the .gitignore. In that file are things like terraform .tfstate and the .terraform directory. Those shouldn't be committed to source control. I'll run git add and then a period to add all files and folders in the current working directory, except for the ones matching gitignore entries and then run git commit m First commit to commit the files to the local repository. Heading back to Azure DevOps, I'll go to the repo service and there's instructions there for pushing an existing repository. Perfect. I'll copy that command and paste it over in the terminal. Essentially, we're linking our local repository to the remote one hosted in repos. The second command pushes our

local main branch up to the remote repository at which point I might be prompted for credentials. Since I've done this fairly recently, my credentials are still cached locally, so I was not prompted, but if I had been, I would have to select the correct identity, and after a few moments, the code would be pushed. Heading back over to Azure DevOps, if your IDE doesn't support the integrated dialogue for provisioning credentials, you can generate credentials using the link under clone to your computer and then add them as Git credentials. Clicking on the Files area, sure enough, here's our files from the repository. Excellent. Before we actually start incorporating the pipelines, there's a few other things we need to take care of starting with state storage. If we're going to be using Azure DevOps pipelines to automate our deployment, we need to move Terraform state to a remote location. I'm not going to rehash the entire process since we went over it in a previous section, and there's a whole lab for setting it up. Instead, we'll start with a storage account I've already provisioned. I've given my own account access, but we also need to give the service principal used by the pipelines access as well. I still have the tab open with the name of the service principal, so I'll grab that and head over to the IAM section of the storage account. From there, I can add a new role assignment, and out of the list, I'll look for the storage blob data contributor which will be enough rights to write and read state data. On the next page, I'll assign the role to the service principal and then review and complete the assignments. Now, we just need to migrate our state up to the storage account. Back in VS Code, I'll open the terraform.tf file and add a backend block setting the type to Azure RM. I'm not going to put any other configuration information in here because I want this configuration to be reusable and not assume the name of the storage account, container, or key. To migrate the state data, I'll pull up the terminal and run Terraform init. I have the command stored here in a text file so I don't screw it up. I'll use the backend config flag multiple times for each argument we need to add to the config like resource group name, storage account, container name, and key name. Once I kick off the command, Terraform will ask if I want to migrate my state data over to the new back end. I'll confirm and now the state data has been successfully migrated to a remote location that the pipeline will have access to. The last thing we need to set up for our pipeline is variable values, so let's talk about some options for configuring those.

Defining Input Values

There are a lot of ways to set input variable values in Terraform, but not all of them are well suited for an automation scenario, and it doesn't answer the larger question of where to store those values for the automation process to retrieve. Let's cover the high level options and why you might want to choose one over the other. There are four basic ways that you can submit a value for an input variable in Terraform. You can do it through a specifically named file, through an environment variable, at the command line with the var or

var file arguments, or when prompted after running Terraform. With the exception of being prompted at the command line, which isn't really an option for an automation scenario, all of these other options will work, but you still need to get the configuration values from somewhere, so where do you get them? You could simply store the values with your code like we've already been doing with terraform.tfvars. Check that into source control with everything else and let Terraform find it during the run. That works, but it doesn't really support multiple environments easily. Alternatively, you could store the values for each environment or deployment as a separate tfvars file checked into source control and then have your automation pipeline select the correct file based on some internal pipeline setting. There are a few issues with storing your configuration data with your code. It's static by nature, so it doesn't update dynamically based on external changes, it's readable by anyone with access to the repository, which isn't great for sensitive data, and any updates would need to go through a code update process which may or may not be useful. If you're looking for something more dynamic, able to store sensitive info, or simply stored separate from your code, Azure has some options for you. Azure DevOps pipelines has a library feature where you can create variable groups and grant pipelines access to read the contents of those variables. This is great for when you want to make your pipelines reusable, and it can also be used to store input variable values for Terraform. Outside of pipelines, you can store your configuration data in an Azure service like Key Vault or AppConfig. Key Vault is great for sensitive information like a VM password or an API key, and you can integrate it directly with the pipeline as a task or using a variable group. AppConfig lets you store arbitrary key value pairs and files. It can be linked to Key Vault to store sensitive information as well. You can pull the value stored in AppConfig through the pipeline and use them as input variable values. You can also use non Azure services like HashiCorp Vault or console or your own homegrown service. Lastly, you can skip the use of input variables altogether and use data sources to pull configuration information from wherever you want, but you'd still need at least some input variables to tell Terraform about those data sources. You can target services like Azure DNS, app_configuration_keys, or key_vault_secrets all from within your configuration instead of in the pipeline. Another possible data source is terraform_remote_state which exposes the outputs of another configuration's state data. HashiCorp generally recommends against doing this because the target configuration needs to have full read access to the state data, but it is an option if you need it. I just presented you with a bunch of different options for storing your configuration data, but I want to make two important things clear. Number one, you do not have to choose a single solution. You could use tfvar files to store values that are relatively static, not sensitive, or not tied to an external resource, and then use data sources in Key Vault for other values. There's also no one right way. Your organization probably already has an established place to store configuration data and it may differ from what I show in the demo. What's more important here is to understand the options, why you might want to choose one over the other, and where they fit

into the automation process. That being said, let's get into a demonstration.

Demo: Adding Azure Key Vault and App Config

Globomantics likes to store their application configuration data in AppConfig, and they don't think Terraform should be any exception. They'd like us to use an AppConfig instance to store our input variable values, instead of the existing Terraform.tf var's file. They would also like us to store sensitive information in key vault. In particular, there will be an API key for the Taco Wagon app that needs to be loaded on each machine. In the demonstration, first, we will add values to an AppConfig resource I have provisioned. Then we will add the API key to Azure Key Vault for safe storage. With our configuration data ready to go, we'll build out two pipelines, one to handle pull requests and run a plan and another to handle applying the plan when the pull request is approved and merged. Then we'll run the pipelines using a GitOps workflow to introduce a change and confirm that our pipelines work properly. Let's get started. Here in the Azure portal, I've already provisioned an AppConfig instance and a key vault. If we go into the IAM role assignment for AppConfig, I have already granted the pipeline service principal access to read the values in this AppConfig instance. Now we can populate the key value pairs. In the AppConfig instance under Configuration explorer, I'll create a key called subnet_id and then go back to VS Code and copy the existing value from our current configuration and paste it in for the value. I'll leave the label and content type blank. Then I'll create a second key value pair with the key being application port and set the value to 8080. The next thing to set up is a key vault entry for our API key. AppConfig can directly reference an entry in key vault, and I've already stored a secret in key vault called API key. I'll add a new entry, but this time select a key vault reference and then pick the corresponding key vault and secret from that key vault. Now, we only need to integrate AppConfig into our pipeline and not both services, that's pretty convenient I'd say. Before we run the configuration, we'll have to add the API key as an input variable and update our user data script, but first, we need to create some pipelines. Azure DevOps can store the deployment pipelines along with the rest of the code in repos, so we are going to build out our pipelines in YAML and then add them to the pipeline's interface. We are going to create two pipelines. The first will run when a new PR is created targeting the main branch. I have that pipeline already written in VS Code, so let's take a look at it. In my directory app vms update, I have a directory called pipelines, and in there is a file called pr plan.yaml. That is the pipeline, and if we open it up, at the beginning, it is loading two variable groups which we'll create in a moment. The first group contains the values for storing our state data, things like the resource group name and storage account name. The second group contains the values for where we want to retrieve our configuration data from, which for now, will just be the AppConfig endpoint. We also have a standalone variable that defines the name of the service connection we will use in the pipeline. I've set it to

tacowagon app to match the name of our service connection. This pipeline should trigger on any commit that isn't in the main branch, so the trigger excludes only main. For the stages, we'll check to make sure the build reason is a pull request. Speaking of stages, there are two stages in this pipeline. The first validates the code using terraform fmt and validate, which means we should first install Terraform. Inside the jobs portion, we are going to use a hosted ubuntu pull. The first step in the job installs the latest version of Terraform using a custom task called TerraformInstaller. Then there's a series of script tasks that run our validation process. The TerraformInstaller and other custom Terraform tasks are part of the Terraform extension from the Azure DevOps Marketplace. I already have the extension installed from previous projects, but I thought I would mention it for completeness. The first script line runs Terraform fmt check, which will error out if the code is not formatted properly, then we will run terraform init to download the provider plugins and modules. The backend false flag tells Terraform to not initialize the back end since we don't really need that for the validate command, and then the last command runs terraform validate. Assuming everything in this stage passes, the pipeline will move on to the planning stage. We'll use the same pull type for the job, and we'll start by getting our values from AppConfig and saving them as a file. The task type is AzureCLI and we have an inline script that will run. The script will export the key value pairs it finds at the endpoint stored in the variable AppConfig endpoint, which it gets from the configuration data variable group. The destination will be a file formatted in JSON at the system's working directory called terraform.tfvars.json. With our configuration values saved to a file, we can install the latest version of Terraform and run a plan. Of course, first, we need to initialize Terraform for real this time, and we'll do that with the TerraformTask custom task and pass it some information. The command will be init, and the backendServiceArm refers to the service connection name to use for which we are referencing our service connection variable. The rest of the settings configure the back end arguments to use during Terraform Init, and they all come from the state data variable group. Under env, we are setting the ARM_USE_AZUREAD and ARM_USE_OIDC to true to make sure the authentication works properly against the storage account, and we're also setting some Terraform specific environment variables to enhance logging and skip any input prompts. The next task uses the plan command with an additional option to save the plan out to a file, deploy.tfplan. The argument environmentServiceNameAzureRM will be the service connection again, and we are setting all the same environment values. This task will produce a tfplan file that can be used when we merge the pull request and run the terraform apply, but what do we do with the file in the meantime? Ah, we can use the published pipeline artifact task to save it as an artifact for use later. That's it for the PR pipeline. Let's set up those variable groups and update our config so we can see it run. Over in Azure DevOps pipelines, I'll create the first variable group called state data and add the entries here for the Azure RM back end we'll be using. These will match what we used locally to migrate our state data. And after a few

minutes, there we go, now, we'll create a second variable group and call it config data, and in there, I'll add one entry called app config endpoint with the endpoint of the AppConfig instance. Alright, that looks good. Now it's time to update the configuration. Back in Visual Studio Code, I will start by creating a new branch for our configuration called api key. This is our feature branch to test our changes. We're adding an API key, so in the variables.tf file, I'll add a new input variable called API key and set the type to string. To show it's being loaded properly, I'll add it as a value being used by the user data template file with the API key set to the var.api key. And then over in the template file itself, I'll add a section of text that interpolates the API key value that we submit to the template file function. Lastly, I'm going to copy the pipelines folder out of the updates directory and into the app vmss directory. With all of that in place, I'll add my changes to the local repository with git add, then commit the changes with git commit and a message about adding the api key, and then push the changes up to the remote repository, creating the corresponding branch using git push set upstream origin API key. With our code now uploaded to the remote repository, I'm going to add the PR pipeline. Under Pipelines, I'll create a new pipeline and select Azure Repos Git and then the repository in this project. I'll pick the existing pipeline file from the drop down and pick the api key branch to grab it from. It will look for any YAML files found in the repository, and there's our PR pipeline. I'll pick that and then click on Continue. We'll now get a preview of the pipeline code which I can choose to save rather than run. Once the pipeline is created, it uses the name of the repository by default, which isn't very helpful, so I'll click on the three dots and choose to rename the pipeline to pr plan. Okay, while we're here, I'm also going to add the merge apply pipeline, although we haven't looked at it yet. We'll get to that in a moment. Once we finish adding it, I'll rename it as well to merge apply. And now we have our pipelines. To get the PR pipeline to fire when a pull request is created, we have to add a build validation policy to the main branch, that is done by going to the Branches portion of repos and clicking on the three dots and then Branch policies. Under there, I'll click on Build Validation and select the pr plan pipeline. This ensures that when a pull request is created, this pipeline will trigger and that you cannot commit directly to the main branch. Now we're ready to create our pull request by going to the pull request portion of Azure Repos. It will prompt us to create a PR for the branch we just checked in which is what we want, so I'll create that pull request, and then we can head over to pipelines and watch our pr plan pipeline run. Over in the pipelines area, the PR pipeline is running, so I'll check and make sure the stages pass. It looks like some permissions are needed. Our pipeline is using variable groups which, by default, are not accessible to the pipeline, so I'll grant access and let the pipeline continue. Now, you might have noticed that I didn't format my Terraform code properly when I added the API key variable, so the validate stage is going to fail, and we can go into the details and see that it failed on the format task. To fix that, we can jump back to our code and run a terraform fmt to fix it, then commit and push our changes back up to the remote repository. That will kick off another run, and this time, hopefully it will

succeed. Let's fast forward to when it's completed. Well, it did succeed, but I actually had to run it twice because I forgot to grant permissions to the key vault for the service accounts, so I had to kick off the plan job a second time, which is why there's two attempts. If we expand attempt number 2, that one was successful, and if we want to see the changes that are going to be made, we can go down to the Terraform Plan stage, and within there, it will list out all the proposed changes. I'll scroll down so we can see what those changes are. It looks like it's going to update our virtual machine scaleset user data in place, which seems correct, but before we merge the change, let's take a look at the merge apply pipeline. The merge apply pipeline only needs the state backend config, so we only add that variable group. For the trigger, it should only run on a commit to main, which happens when APR is merged. Under the stages, things are pretty simple. First, we use the DownloadPipelineArtifact task to grab the plan file from the PR plan pipeline. Then we install an initialized Terraform in the same way we did for the previous pipeline. Then the last task is to run apply with the tfplan file added as a command option. As long as the planned file is valid, Terraform will run the apply automatically, so let's do that. Heading back over to Azure DevOps, I'll merge the pull request which should kick off the merge apply pipeline. Sure enough, looking over in pipelines, it is running, but once again, we have to grant permissions for it to access the variable group. I'll do that and then the pipeline will get started, and then through the magic of editing, the pipeline completes successfully and our environment has been updated. I'll go into the Terraform Apply task, and down at the bottom, I'll grab the IP address from the output and then open up a new tab and browse to that IP address, and after a few moments, there's our web page with our API key proving that it loaded successfully. Sweet. In this demonstration, we have automated our Terraform deployment with Azure DevOps, AppConfig, and Key Vault. There's certainly more that could be added to the process with automated tests and multiple environments, but this is a great start.

Going Cloud Native with Kubernetes (AKS)

Going Cloud Native with Kubernetes

While traditional compute like Azure VMs form the bedrock for applications and platforms, I think I'd be remiss if I didn't also talk about cloud native technologies like containerization, and more specifically, Azure Kubernetes Service. Managed Kubernetes Services are wildly popular in the public cloud in part because Kubernetes itself is very complex and difficult to administer, so let's take a look at what container technologies exist on Azure and how to manage AKS with Terraform. We will start the section with a review of the various container services available on Azure and then talk specifically about the Azure

Kubernetes Service. We'll get into the networking behind an AKS cluster, the security options, and management lifecycle. Then we'll take a look at the AKS preferred architecture from Microsoft and how to use the AKS module from the public registry to deploy an AKS cluster that follows the preferred architecture. With our AKS cluster in place, we'll deploy Flux CD using Terraform and then use the Flux installation to deploy an application to our cluster. First, let's talk about container technologies in Azure and AKS, in particular.

An Azure Kubernetes Service Overview

When you think of cloud native technologies, the first thing that jumps to your mind is probably Kubernetes, and that's fair. It's certainly the most successful project to come out of the cloud native computing foundation, but there are many, many ways to leverage containers to run applications on Azure and other platforms. To a certain degree, Kubernetes is an orchestration engine whose primary job is to manage the lifecycle of container based workloads, but it's not the only option by far. Before we go too far down the AKS rabbit hole, I'd like to mention some other options. Azure Container Instances allow you to run a single container or a group of containers without provisioning any other infrastructure. You don't need to create a virtual network, an Azure VM, or a storage account. It is by far the simplest way to run a container on Azure, but it also doesn't include scaling, load balancing, or certificate management. Azure Container Apps is a more robust solution over ACI and is built on top of Kubernetes. It allows you to run Kubernetes style applications without having direct access to the underlying Kubernetes APIs. If you want to run a Kubernetes style application and don't need access to the underlying constructs, this could be a solid option. Azure App Service is used to host web applications and web APIs. You can supply Azure App Service with web application code or use containers as a source. App Service is really geared towards web applications which might not fit your use case. Azure Functions is Microsoft's flavor of functions as a service, and you can use either code or a container image as a source for execution. If you're building a function as a service based application component, this could be a good option. There's also Azure Red Hat OpenShift which is a managed version of OpenShift jointly engineered and operated by Microsoft and Red Hat. OpenShift is Red Hat's implementation of Kubernetes with a bunch of bells and whistles added. And let's not forget about Service Fabric, a platform for running microservice applications that may or may not use containers in the background. Microsoft uses Service Fabric extensively for their own services, and while it's not a container platform per se, there are definitely containers on the back end. That's not even all the options for running containers on Azure, but I think it's a solid overview. For the remainder of this section, we'll be focused on Azure Kubernetes Service, or AKS. Azure Kubernetes Service provides a managed Kubernetes cluster experience made up of several key components, some of which are managed by Microsoft and some of which are

your responsibility. AKS uses existing core primitives in Azure to build the service, including Azure VMs, scale sets, virtual networks, load balancers, and application gateways, and Entra ID for authentication and authorization. When you provision an AKS cluster, Microsoft deploys an abstracted control plane that you are not charged for. The workloads you want to run on AKS will be hosted on nodes in a node pool. Typically, these nodes are made up of virtual machine scale sets, although it is possible to burst to ACI using the virtual Kubelet. A single AKS cluster can have a mix of node pools, including both Windows and Linux based pools. Node pools can be scaled horizontally as needed using the cluster autoscaler. The node pools you provision will live in a virtual network, either one dedicated to the AKS cluster or an existing VNet with other services. AKS is also integrated with network services like internal or public facing load balancers and the application gateway. For persistent storage, AKS can leverage Azure disks or Azure files depending on the use case, and AKS leverages both Kubernetes RBAC and Entra ID at the node and pod level, meaning that users can gain access to the cluster through Entra ID authentication and RBAC, and that nodes and pools in the cluster can use Entra ID to authenticate out and gain access to other Azure services. Since AKS relies heavily on networking in Azure, we should probably spend a few minutes reviewing the integrations and choices you'll need to make when deploying an AKS cluster. These decisions will inform what resources you need Terraform to provision before it creates the AKS cluster. Like I mentioned a moment ago, AKS relies on Azure virtual networks to host its node pools. There are essentially two networking options when it comes to AKS. You can use Kubenet or Azure Container Networking Interface. Kubenet is the simpler of the two deployment options, and it's the default if you're deploying AKS through the portal. Every node in the cluster will get an IP address from the virtual network space, and the pods will get their IP addresses from a separate pool on each node and leverage NAT to get to the resources on the virtual network. Microsoft's recommendation is to avoid Kubenet for production deployments. Instead, you should select CNI, or the Container Networking Interface. There are a few CNI flavors depending on your cluster requirements. The original is the Azure CNI, also called Advanced Networking. With Azure CNI, every pod gets an IP address from the subnet it was deployed in. The pool of IP addresses is pre allocated to the nodes in your pool, which can lead to address exhaustion if you're not careful. Proper planning is critical for a well functioning AKS cluster. A newer option is Azure CNI with dynamic allocation which allocates IP addresses to each pod from a pod subnet, instead of from the nodes directly. It's a slightly more complicated setup, but it also provides more flexibility for environments that may scale unexpectedly. Rather than allocating an IP address from the VNet for each pod, Microsoft has introduced Azure CNI overlay networking. This option creates an overlay network with a private IP address space that the nodes and pods can use to communicate. Azure CNI overlay is the recommended option from Microsoft, but it doesn't support some ingress controllers. If you're planning to implement network policy enforcement for AKS, you can use either Azure network policy or

Cilium with the Azure CNI. Cilium is the preferred solution for larger deployments or when you want a consistent solution across cloud providers. And finally, there's bring your own CNI. If you love Calico, or Flannel, or Weave Net, you can use any of those for your CNI, but bear in mind the management of the CNI now becomes your problem, instead of Microsoft's. When it comes to security in AKS, there's a few different layers to consider. First, you have the authorization and authentication for the Azure resources that make up your AKS cluster. Then you have to consider the security of the cluster resources like nodes, deployments, and service accounts. And finally, there's the security of the network and the data that's being transmitted. For the AKS resources, you'll be relying on Entra ID for authentication and Azure RBAC for authorization. There are built in roles available for AKS or you can define your own custom roles to fit your needs. For workloads running on the cluster and the cluster itself, you can rely on the internal authentication of the cluster or you can integrate with Entra ID. Microsoft's recommendation is to use Entra ID for authentication. For the pods, in particular, you can choose to go with a managed identity or workload identity. For authorization, you can use a combination of built in Kubernetes RBAC to control access to the cluster and Azure RBAC, which can be extended to grant access to the Kubernetes API and resources inside of the cluster through roles and cluster roles. For the network and data security, you can secure traffic using Azure Firewall and network security groups and use Azure Network Policy to control the flow of traffic to and from the cluster. By default, AKS clusters have a public endpoint for access to the Kubernetes API. You can lock down access to the public endpoint using firewall rules, but if you'd like to keep all the API traffic off the internet, you can use the private cluster feature to publish the API endpoint through a private link. There are two primary components in AKS that need to be updated on a regular basis, and you can control the update frequency and lifecycle for both of them. The first component is the node image running on the nodes in the node pools of your cluster. Microsoft releases nightly security updates for the operating system on Linux based nodes and weekly updates for both operating system types. You can control when the update and images are applied to your nodes, including specifying a maintenance window for the updates. The second component is the version of Kubernetes itself which is updated on a quarterly cadence. You can control what version of Kubernetes your cluster is running and when updates are applied. Microsoft provides full support for the latest and last two minor versions of Kubernetes. If you need to stay on an older version of Kubernetes for an extended period, Microsoft does offer an LTS option that extends the support window for an additional 2 years. There are a lot of considerations to take into account when updating your nodes or clusters, so it's important to review the documentation and plan accordingly. AKS is just a managed version of Kubernetes, so you can deploy applications to it in the same way that you would deploy applications to any other Kubernetes cluster. You can use kubectl, Helm, or any other Kubernetes compatible tool to deploy applications to your AKS cluster. Microsoft does have a few other ways to deploy applications to AKS that are more native to the

platform. You can install flux as an AKS extension. AKS extensions are maintained by Microsoft and installed using the Azure Resource Manager API. Flux is a GitOps operator for Kubernetes and it can be used to deploy applications to your AKS cluster. You can also use the Azure Marketplace to deploy third party applications to your AKS cluster. The Azure Marketplace has a number of applications that are preconfigured to run on AKS, and you can deploy them with just a few clicks. The marketplace items also use the ARM API to deploy the application. In either case, if a solution uses the ARM API, it can also be deployed using Terraform as we'll see later. But first, let's build out an AKS cluster using Terraform.

Demo: Creating an AKS Cluster

Globomantics has decided to run some of their Taco Wagon app on AKS. They've asked us to build out an AKS cluster that follows the preferred architecture from Microsoft. Let's take a look at the preferred architecture and then build out the AKS cluster using Terraform. We'll start the demo by reviewing the preferred architecture for AKS and using it as a template for building out the Terraform configuration. Then we'll take a look at the AKS module on the public registry and see how we can use it to build out our AKS cluster. Finally, we'll build out the Terraform configuration and deploy the AKS cluster. Let's get started. Microsoft has published a baseline architecture for AKS that they recommend for most production workloads. We are going to reproduce portions of this architecture using Terraform. The preferred architecture includes a hub virtual network containing network security components like Azure Firewall, Azure Bastion, and a gateway subnet. The Hub network is connected to one or more spoke networks that each contain AKS clusters and other resources. We are going to focus on the spoke network component of the preferred architecture. The spoke network will contain several subnets for Private Link, ingress resources, and the Application Gateway, as well as for the AKS cluster itself. To improve the security of images and store secrets for the cluster, we will also deploy an Azure Container Registry and Key Vault that are connected to the spoke network using Private Link. The AKS cluster will be able to use the Private Link along with the machine and workload identities to access the Azure Container Registry and Key Vault. The cluster will be composed of two node pools, one for system resources and one for user workloads. Both node pools will be running Linux and leverage VM Scale Sets. For authentication and authorization out of the cluster, we will use Entra ID and machine and workload identities associated with the AKS cluster in Kubelet, along with Azure RBAC to allow the cluster to access resources like Azure Key Vault, Network resources, and the Azure Container Registry. We can also leverage workload identity to provide workloads inside the cluster access to Azure resources. To allow access into the cluster, we will use Entra ID and Azure RBAC to control access to the Kubernetes API and resources inside of the cluster. This includes disabling local accounts since we won't need them. For

internal networking in the AKS cluster, we will use the Azure CNI networking option, and for network ingress, we will use a combination of the Application Gateway Ingress Controller and internal load balancer. This is by no means an exhaustive look at all the components that could go into an AKS design, but it's a good starting point for a production-ready AKS cluster. With this architecture in mind, let's take a look at the AKS module on the public registry to see what's already included with the module. This is the AKS module on the Terraform public registry, and the first thing I want to note is that there are 147 possible inputs. We are not going to review each of these in exhaustive detail. As a useful starting point, the module has several examples for common use cases. Once you select an example, I'll pick the ACR one, it will take you to a new page that shows a read me for that example. To view the actual code, you'll need to go to the GitHub page and check out the main.tf file. All of the examples are here in the examples directory, and chances are that none of them are going to exactly fit your use case. Instead, you can use them as a jumping off point for your own configuration rather than starting from scratch, at least that's what I did. Let's check out the configuration I built with this module and some other helpful modules from the public registry. Heading over to VS Code, this configuration is pretty big, so for simplicity, I broke it into several files. That makes it easier to find the relevant code for the different components. I've got networking.tf for the networking, aks.tf for the AKS cluster, etc. For the networking, we need a spoke VNet that has the relevant subnets laid out. In the networking.tf file, we are using our good friend, the vnet module, as well as creating a resource group specifically for networking. The subnet prefixes and names are coming from an input variable, the value for which is in terraform.tfvars. As we saw in the diagram, we've got subnets for the Private Link, load balancer ingress, Application Gateway, and node pools. That takes care of networking. Now let's get into the AKS cluster by opening the aks.tf file. We are creating a separate resource group for the AKS cluster, which I'll also use for Key Vault and the Container Registry. Inside the AKS module block, I've chosen to organize my input variable values by function, instead of alphabetically. When I want to adjust something later, I can quickly find the relevant input variable by skimming the sections. For instance, we want to use the azure CNI plug-in and calico for network policy. Those two settings and the subnet to use for the system node pool are under cluster networking. When working with large complex modules like the AKS module, keeping your inputs organized is pretty helpful. Heading down a little bit, we have our workload node pool defined using the same subnet as the system pool. You may decide to locate the node pools on different subnets, depending on isolation requirements and IP address planning. Let's get into authentication and authorization. We want to use Azure RBAC and Entra ID authentication. That means the local account on the Kubernetes cluster should be disabled. We also need to enable a bunch of other settings. RBAC AAD adds Entra ID integration. Rbac_aad_azure_rbac_enabled configures role-based access control through Entra ID, and rbac_aad_managed uses an Azure-managed service principal for the integration. Using Entra ID integrated RBAC requires that

the role based access control be enabled on the cluster itself. Because we want to use workload identity as well, we also need to enable the OIDC issuer. Just looking at these input variables might not be all that helpful, which is why it's so important to read the documentation for the module and the Microsoft docs for AKS, in general. There is a few other arguments to cover in the add on section. We're integrating with Key Vault, so the Key Vault services provider extension is enabled. We're using Azure Container Registry, so the attached_acr_ip_map argument will set up proper permissions that allow the cluster's managed identity to pull images from the registry. And lastly, we're using an AKS managed App Gateway for ingress, also known as a Greenfield application gateway for ingress, and that needs the subnet ID where the App Gateway should be hosted. The last block here grants my account several roles to manage the Kubernetes cluster, otherwise I wouldn't be able to monitor, manage, or deploy the cluster. That's it for the AKS configuration. Now I want to briefly touch on the Key Vault and Container Registry files before we deploy this behemoth. For the Key Vault, I decided to use the Azure verified module for Key Vault. I mentioned these verified modules earlier, and I decided to use the Key Vault one because it has built in support for Private Link, something we'll need to let our AKS cluster access Key Vault. Better yet, there's a great example for doing exactly that in the module repository. For the most part, I grabbed this example and pasted it into my configuration and then made a few slight changes. We are creating a private DNS zone for the Vault Core Private Link address. Then in the module, we are creating a private endpoint using the Private Link subnet and DNS zone. The public network access is set to false, so this Key Vault is only accessible through Private Link. We also want to allow the AKS cluster to use either its Key Vault integration or workload identity to access Key Vault. There is two role assignments granting Key Vault administrator to the integration identity and a user managed identity created for workload identity integration. Lastly, as part of the integration, each workload that wants to use the workload identity will need a federated credential with the AKS cluster's OIDC service as the issuer, and the subject mapping to the namespace and the service account of the application. I've set up a test one for the default namespace and a testing service account. Over in the ACR file, I'm also using one of the verified modules, and the consistency makes it easy to set up this portion too. We create a DNS zone for Private Link, and then the private endpoints argument for this module works exactly the same as the Key Vault module, taking the DNS zone and subnet ID as arguments. And just like that, we have a configuration that maps our desired architecture while leveraging modules to make the configuration simpler. Let's go ahead and deploy this bad boy. I'll pull up the terminal, and I've already initialized the configuration, so I'll simply kick off a terraform apply with auto approve. The actual deployment will take a significant amount of time, so while we're watching it go, I wanted to mention that while Terraform does make it easier to deploy infrastructure on Azure, it doesn't explain the whys or the hows. The modules are super helpful in the sense that they can have same defaults or streamline a deployment option, but they also don't tell you why. If

you aren't well versed in the underlying Azure services and their various options, then Terraform is not going to be of much help. When I need to set up something new in Azure, I spend 80% of my time reading docs and probably 20% of my time writing Terraform. That's why we started this demo with our desired architecture, rather than just jumping to the code, and I think that should be your approach as well. Okay, through the magic of editing, our AKS cluster has been deployed successfully. I actually had to run the apply a second time because the name of the Key Vault I wanted to use had already been taken, but everything has been deployed, so we can head over to the portal and confirm exactly that. There's our AKS cluster, and it has the node pools we wanted, and the networking is set up appropriately with Azure CNI and Calico, and the app Gateway Ingress Controller. Local accounts are disabled and Entra ID auth with Azure RBAC is configured. However, just like our example with virtual machines, this Kubernetes cluster is feeling a little bit lonely. How about we lay down an application?

Demo: Deploying Applications on AKS

There are a plethora of ways to deploy applications on Kubernetes. You could use kubectl, Helm Charts, or even send YAML manifests via the API server directly. But in the same way that we implemented a declarative GitOps approach for our Terraform code, we can do the same with our AKS cluster by using Flux CD. Globalmantics has decided to use Flux CD for their ongoing management of their AKS clusters and the applications deployed on them. This was informed in part by the fact that Flux is available as an extension on AKS and surfaced up as a network interface in the portal. For our demonstration, we'll start by installing the Flux extension on our existing cluster and then taking a look in the portal. Then, we'll review an example repository that has a Flux configuration we want to deploy. We'll talk a little bit about the basics of using Flux CD and the available Terraform resources. Then we'll build a new Terraform configuration that deploys the Flux config to our AKS cluster and verify that all the components install successfully. But first, let's get that extension installed. Over in VS Code, I have the aks.tf file open from our AKS deployment. Since Flux is a native extension, we can use the Terraform resource, azurerm_kubernetes_cluster_extension to install it on our existing cluster. You can use this resource to install marketplace items as well. For the name, we'll call it fluxext, and the extension type is microsoft.flux. The cluster ID can be sourced from the AKS module from the output aks_id. Now, I'll pull up the terminal and run a quick terraform apply to deploy this extension. This will take about 4 or 5 minutes as Microsoft installs all the extension components on the cluster. Jumping ahead, it has finished deploying, so let's head over to the portal and see what was installed. In our AKS cluster, if we go down to Extensions and applications, there's the Flux extension that has now successfully been provisioned in our cluster. If we go up into namespaces, within the

namespaces, there's now a dedicated namespace for flux system, which will include the pods and services that Flux needs to do the reconciliation. The configurations that will be managed by Flux will appear down in the GitOps section, but right now we haven't loaded any configurations, so why don't we get a configuration installed? Open in another tab, I have a GitHub repository that contains some configurations that we can deploy using Flux. Now Flux is able to synthesize Kubernetes manifests from many different sources. It could be a Git repository like this, OCI or Helm repositories, or even blob storage like an S3 bucket. In our case, we have two different configurations in here, one is to configure the cluster, and the other one is to install an application. The cluster configuration is under the infrastructure directory, and in here is a customization file, which is one of the ways you can deploy a configuration using Flux. Opening up the customization file, the customization is a custom resource definition used by the Flux controller. In this case, we're pointing it at some resources that we want to deploy through this customization which are stored in the source's subdirectory. Inside the source's subdirectory is another customization file, which Flux will find and process. In that customization file, we're simply pointing it at the bitnami.yaml file to process what's in there, but we could point it to multiple files. When Flux processes a declarative configuration, it deploys it in the cluster through a process called reconciliation. Flux not only does the initial reconciliation and deployment, but it also checks back periodically for updates and reconciles the deployments to match those updates. If we look at the contents of the bitnami.yaml file, it contains a kind of Helm repository. Helm repository is another custom resource definition that we can set up using Flux, and this will correspond to the Bitnami Helm repository, and it's going to check on that every 30 minutes to see if it needs to be updated. Using the customization file in the sources directory, we could add additional Helm repositories, or we could add other infrastructure specific configurations by creating more YAML files and referencing them either using the Resources section here or in the Resources section of the parent customization file. That'll do it for the infrastructure configuration. Heading back up to the root of the repository, we have the dotnet directory. This uses a customization file as well to install an ASP.NET application. The customization specifies two resources, the namespace.yaml file and the aspnetapp.yaml file. We also have a patches portion that can overlay the contents of what's in the aspnetapp file. The resources are processed in the order that they appear, so the namespace.yaml file is processed first. This will create a namespace named aspnetapp, and then it will process the contents of the aspnetapp.yaml file. In there, we have a pod, a service, and an ingress defined that will deploy the ASP.NET application. This deployment relies on the presence of the namespace that we create before deploying the application. It is also using an ingress class name of NGINX, but that's not what we're using in our cluster. We're using the Application Gateway ingress, so we need to patch the contents of this configuration and we do that in the customization.yaml file. Inside the customization, we have a patches portion that is patching inline the ingress class name to azure application gateway, so it

will use the Application Gateway as an ingress, instead of NGINX. If we wanted to support multiple environments, we could have subdirectories for each environment and do something like overlay the namespace with staging, prod, or development. Now that we know what we'll be deploying, let's set it up in Terraform. Since Flux is a native solution in Azure and represented by the ARM API, we can use the AzureRM provider and the resource AzureRM Kubernetes Flux configuration to add configurations to our AKS cluster with Terraform. I have a basic configuration here that includes input variables for the GitHub repository URL that holds our Flux config and the resource ID of the AKS cluster. Over in the main.tf file, we have our provider block for the azurerm provider, followed by the resource block for our Flux configuration. For the name, I'm naming it tacopod, and the cluster ID references the input variable cluster_id. The namespace is set to cluster config, and the scope is cluster. To link it to our Git repository, we'll use the git_repository block, and inside of that block, we'll use the url argument and set it to our input variable of our GitHub repository. The reference type, in this case, is branch, and then the reference value will specify which branch it should be tracking. In this case, I have an input variable named flux_git_repo_branch, which has a default value of main. Back in the main file, in order to load our customizations, we use the customization block, and you can specify it multiple times. In this case, we have one customization block, we'll call it infra, and the path to it is ./infrastructure, which is relative to the repository. Under the other customization, we'll call this one aspnetapp, and the path for that will be ./dotnet, and we'll set this to a sync interval of 60 seconds, which means that Flux checks every 60 seconds against the repository to see if anything has changed about the files that are in the dotnet directory. In my terraform.tfvars, I already have the correct GitHub repository URL set, but we need to set the cluster ID. I'll pull up the terminal, and one of the outputs from our AKS deployment is the cluster ID. I'll copy that value and paste it in as the cluster ID. Now we're ready to deploy our Flux configuration. I'll pull up the terminal and move to the correct directory. I'll kick off a terraform apply, and that will deploy the Flux configuration. Once we have the Flux config in place, further updates can happen directly in the GitHub repository that it's tracking. Essentially, Terraform sets up the Flux config and then gets out of the way. Flux is much more suited to actively managing Kubernetes clusters and applications, so it makes sense to leverage its capabilities once the AKS cluster is provisioned. Alright, our Flux configuration has been successfully deployed, so let's head over to the portal and take a look at what has been configured there. Back in the portal, I'll refresh the view, and now we see that our tacopod configuration is compliant and succeeded, meaning that it both applied the current configuration and the created items match the desired state. Clicking into the configuration, there are four objects expressed in the config, and those objects are the GitHub repository that contains our code, the customizations we defined, and the Helm repository that was created by the infrastructure customization. Our aspnetapp customization should have deployed an aspnetapp pod service and ingress, so heading over to the Ingress's section of our AKS cluster, there is one ingress using

the Application Gateway. That means our customization replaced the NGINX default value with the Application Gateway used by our cluster. Clicking on the link for the ingress takes us to the .NET sample application page, which means that our app was deployed successfully using Flux. When the application team at Globomantics wants to patch or update their application, they can do so by making updates to the GitHub repository, and after about 60 seconds, Flux will pull the update and apply it to the application. This GitOps thing is pretty neat.

The Resource That Wasn't (AzAPI)

Using the AzAPI Provider

If you've worked with Azure and Terraform for a little while now, you already know that the AzureRM provider doesn't always have complete feature parody with the Azure Resource Manager API. In this section, we'll see how you can work around that limitation and how to remove the workaround once the provider is updated. First, we'll dig into the AzAPI provider, the background for its creation and the common use cases. Then we'll take a look at the resources and data sources that exist in the AzAPI provider and how to use them. Finally, we'll take a look at a few AzAPI examples and use them to deploy parts of a solution for Globomantics. But first, let's talk about why the AzAPI provider exists in the first place.

Why AzAPI?

The services and features offered by Microsoft Azure are constantly evolving and changing, including preview and beta features that might not ever hit general availability. To help solve this challenge, the Terraform team at Microsoft created the AzAPI provider. To understand the AzAPI provider, it helps to understand a little bit about how Azure works. Under the covers, the Azure API is broken up into various individual APIs and resource providers. The management engine behind all those resource providers is the Azure Resource Manager. You ever wonder why the provider for Azure is called Azure RM and not just Azure? Now, that's because before Azure Resource Manager or ARM existed, Azure used a solution called Service Manager, and the Terraform provider for that was called Azure. So when it came time to create a Terraform provider for the Resource Manager, they called it AzureRM. Just a fun bit of history in case you've ever wanted to know. Whenever a new service or feature launches in Azure, it has a corresponding API provider and support from the Azure Resource Manager template, or ARM template as they're commonly known. The ARM template is really a thin abstraction over the API, so there's nothing Microsoft has to do to add support for new features or services aside from exposing the correct API endpoint. The AzureRM provider, on the other hand, is a richer abstraction of the

ARM API, and as such, requires someone to update the provider when new features or services are added. The AzureRM provider is an official HashiCorp provider, meaning that there is a dedicated team from both HashiCorp and Microsoft there to address issues, update existing resources, and add new resources as they come available, but that requires a certain amount of effort, and teams need to prioritize which issues and features to support in their pipeline. Anything that becomes generally available will eventually be supported by the AzureRM provider, but preview features and services will probably not. If you need to support a preview service or add a feature that is not yet in the AzureRM provider, what can you do? The original solution for this was to use the AzureRM deployment resources. They let you deploy an ARM template with the AzureRM provider, but they were limited in terms of lifecycle. Terraform wasn't managing the resources deployed by the ARM template, it was only managing the template itself, that meant if you wanted information out of the template resource, you would have to define that information inside of the ARM template outputs, and so Microsoft created the AzAPI provider to solve this exact problem. The AzAPI provider is a thin abstraction over the ARM API with support for any resource that is defined by the ARM API provider, and you can easily migrate to an AzureRM resource if support gets added later. The bottom line is that if you need to interact with services or features in Azure that are not yet supported by the AzureRM provider, the AzAPI provider is exactly what you need.

AzAPI Usage

How does this mystical AzAPI provider work? Let's take a look, starting with authentication and then the resources and data sources included in the provider. The authentication for the AzAPI provider is exactly the same as the AzureRM provider. All the same methods and environment variables will work for the AzAPI, which is a relief. No need to learn a new authentication method to use this provider. Now onto the resources and data sources. As of right now, there are four resources in the AzAPI provider. If that doesn't seem like a lot, it's because most of the magic happens inside of each resource. The AzAPI resource creates an instance of any supported resource from the ARM API, and it's the one you'll probably use most often. However, sometimes there's a resource that is already supported by the AzureRM provider, but there's a feature that isn't. Usually it's a preview feature that might never make it to GA or it's a GA feature that the AzureRM team hasn't gotten around to adding. Either way, there's the azapi_update_resource. This resource type lets you update an existing resource with arguments that aren't part of the AzureRM resource. Next, we have an interesting one that deals with kicking off certain actions in Azure. It's a way to imperatively fire off an action after a resource is created, for instance, having the Azure Image Builder create an image after you create a template with Terraform. Azure doesn't just do that on its own. You have to tell it to do so, and

the AzAPI resource action resource does exactly that. Lastly, we have sort of a companion to the azapi_resource, but for the data plane of Azure. Some data plane resources, like Azure Key Vault Secrets, already have support in the AzureRM provider, but for those that don't, the azapi_data_plane_resource let's you interact with those objects that are not strictly part of the management plane in Azure, for example, loading a data set into Azure synapse, something you might have previously done with a post deployment script. There are also four data sources in the provider. The first two, the azapi_resource and azapi_resource_action are the read only variant of their resource counterparts. For the azapi_resource_action data source, Microsoft recommends using the data source for read only actions and the regular resource for actions that might alter the state of a resource. The other two data sources are really useful utilities for dealing with Azure, in general. The first is azapi_resource_id, which essentially takes the resource ID string and breaks it into its components. The resource id is a really long string that includes the subscription ID, resource group, and other parent resource information. You could try to break it up with a native function in Terraform, but this data source gives you named keys to make things a lot easier. The other one is azapi_resource_list, which will give you a list of all resources of a specific type under a scope. Do you want to get all the Azure functions in a subscription or every subnet in a virtual network? This data source can do that for you. This data source could be especially useful in a governance or reporting module. What started as a provider to handle the specific use case of this solution or feature doesn't exist in AzureRM yet has turned into a bit of a Swiss army knife when it comes to dealing with Azure, but how do they work? Let's take a closer look at the azapi_resource for reference. Inside of the azapi_resource, the resource type is azapi_resource. Inside of the configuration block, we have the name and the parent_id arguments. The name is pretty self explanatory. The parent_id is typically the resource group id, although it is dependent on the resource type. Speaking of type, the type argument is required and corresponds to the ARM provider endpoint and the version of the API, more on that later. The last required argument is the body argument which takes a JSON encoded request body. Remember how I said that this was a pretty thin abstraction over the API. Here's what I mean. The body argument is essentially the request body you would send to the API to manage the resource. There are several optional arguments as well like an identity block if the resource supports assigning a managed identity or a tags argument to apply metadata tags to the resource, but to come back to that parent ID, the type and the body argument, where do you get that information? Well, to the documentation, of course. Over in the documentation for the ARM templates, Microsoft has added a tab specifically for the AzAPI provider, in addition to Bicep and ARM templates. For instance, let's take a look at the Azure storage accounts under the reference/storage heading. The deployment language selection at the top lets us pick Terraform as an option which will show the documentation for the AzAPI provider. We can also pick which version of the API we want to interact with, which I'll just leave that as

latest for the moment. Below that is the resource format which includes the type and a body argument with every possible field you can include. Below the format is an explanation of each field along with the value type for that field. This is not nearly as friendly as the regular AzureRM resource, but then again, it's not meant to be. For other resource types like the data plane resource or the resource action, you'll need to reference the more general Azure REST API docs. These docs don't have an AzAPI specific implementation, so you'll need to do a little interpretation. For instance, if you wanted to add a database artifact to Azure Synapse, we can look under Synapse and expand the data plane reference, and under there is the action Put Artifact in DB. The page lists out the PUT command in the body of the request. It's up to you to convert that into the type parent id and body expected by the AzAPI data plane resource. It is not the easiest way of going about things, but if you need to perform data plane operations with Terraform, there is a way to do it. With all that in mind, why don't we put the AzAPI provider to work for Globomantics.

Demo: Integrate the AzAPI in a Configuration

The primary use case for the AzAPI provider is to deploy a solution or service in Azure that is not yet supported by the AzureRM provider, so let's take the AzAPI provider for a spin. Over at Globomantics, they would like to use the VM Image Builder to create custom images for VM scale sets. Unfortunately, there is no image template resource in the AzureRM provider, so they'd like for you to set one up using the AzAPI. They would also like for you to kick off a build run when the template is updated, which should be possible using the azapi_resource_action resource from the provider. The build processes usually take about 30 minutes, so we'll need to take that into account. Finally, Globomantics used the AzAPI update resource to enable a new retention policy feature for their Container Registries, but that option has now been added to the AzureRM Container Registry resource directly. They'd like your help to remove the AzAPI update resource block without impacting the Container Registry itself. Let's head over to the demo environment and get started. Globomantics wants to use the Azure VM Image Builder to create custom images for the Taco Wagon web application. The Azure VM Image Builder is composed of several components. There's the Azure Compute Gallery, which is what we're seeing here. Inside of that gallery is one or more images. Clicking on one of those images shows that there are multiple image versions. Those versions are created from an image template. The problem is that the image template resource doesn't exist in the Terraform provider. And in fact, according to the maintainers of the provider, it's not going to be added, but it is a resource in the Azure ARM API, so we can use the AzAPI provider to set it up. Over in VS Code, I have a configuration here that sets up the necessary resources for the VM Image Builder. The builder uses a managed identity to provision and run the builder action, so first, we have to create a user_assigned_identity and then assign

it the proper permissions. The first block creates the identity, then the next block creates a custom role with all the necessary permissions, and then the third block assigns the role to the identity. The next two blocks create the Azure Compute Gallery and the shared image that we want to put in that gallery. Part of that definition is the operating system being used and a set of identifiers that a consumer of the image will need to find it composed of the publisher, offer, and sku, that's the same as the marketplace images you might use today. Now, we get into the image template to actually build the image and the syntax of the AzAPI resource block begins with the type which we could defined in the ARM documentation as we saw earlier. We also need to provide a name for the resource and a parent_id and location. The parent_id will be the resource group. Since the builder can't function until the user identity has been created and the role assigned, we've got a depends on that watches for the role assignment to be complete. Speaking of the identity, the AzAPI resource has an optional identity block for associating resources with a managed identity, which we need to do here. Below that, we have the metadata tags to associate with the resource, and then we get into the body argument. This is essentially the request body that will be sent to the Microsoft virtual images resource provider and it needs to be in JSON, but writing JSON is terrible, so instead, we can use the jsonencode function to take regular HCL and convert it to JSON for us. Now, essentially, the VM Image Builder template needs to know what type of VM to use for the build process, which goes in the VM Profile section. then where to get the source image for the build through the source arguments, and then how to customize the image before publishing, which is what we are doing through a series of customized actions. Once the image version is built, we can distribute it to our Azure Compute Gallery using the distribute argument which includes references to the gallery image and regions for replication. Now, even though we've created this template with the AzAPI provider, the builder doesn't actually run until we tell it to, and that is done through another AzAPI resource, the azapi_resource_action. The type for this one will also be image templates, and the resource_id will be the resource we want to execute an action on. The action is run which will kick off the image build process, and the response export values argument gives us a chance to export some or all of the values contained in the API response. I've made this action conditional just in case we don't want to build a new image every time. I've also included a timeout of 60 minutes for the creation process because the image build process can take up to 60 minutes, and we don't want this action to time out because the creation process is taking too long. Let's get this bad boy deployed. I'll pull up the terminal and run a terraform apply, including the -var flag with the build image variable set to true. Like I said, the build process can take up to 60 minutes, so I'll jump ahead to when it completes. Alright, we have successfully built our image. If we head on over to the portal, in the resource group, there is a VM Image definition and image template. Looking at the image template, it has a Build run state of succeeded, and we can see the distribution image below it. Clicking through on that link takes us to the actual image version

which is now available for use by the team at Globomantics. Cool. Now let's take care of the Container Registry issue. Back in VS Code, I have another configuration open for the Container Registry deployment. In the main.tf file, we are creating a Container Registry with some basic settings. Below that block is an azapi_update_resource block that adds the retention policy to our Container Registry. This is a feature that is currently in preview and wasn't available in the Azure RM Container Registry resource until very recently. Just like all the other resources in the AzAPI provider, we need to specify a type, and since we are updating an existing resource, we have to specify the resource_id of the resource we want to patch, and then the body is the request being sent to the Microsoft Container Registry provider, which is really just setting the retention policy to enabled and for 14 days. Now, this configuration has already been deployed. As I mentioned earlier, Globomantics has been using this for a while. Over in the portal, if we take a look at the properties of the Container Registry, under the Retention section, the status is enabled and the retention is 14 days. The retention policy block has recently been added to the Azure RM Container Registry resource, so how do we migrate over? The process is actually really painless. Back in the configuration, in the AzureRM Container Registry resource block, I'm going to add the retention policy block, and I'll set enabled to true and then days to 14 matching what we already had. Then I'm going to delete the azapi_update_resource block, and now, I'll pull up a terminal and run a plan. After a few minutes, the plan is going to come back and let me know that it's going to destroy the azapi_update_resource, but critically, it isn't changing anything about the Container Registry itself. I'll kick off a terraform apply, and after a few minutes, the apply is now complete. If I go back to the portal and refresh the view, nothing has changed about the resource. This works really well for the azapi_update_resource. If you're going to migrate off of an AzAPI resource, instead, you're probably going to need to make use of the import and removed blocks as you'll see in the upcoming lab, but we have taken care of the requests from Globomantics. We've updated the Container Registry and created an image template using the AzAPI resource.

Conclusion

Course Summary

Congratulations. You've reached the end of the Advanced Terraform with Azure course. Before you go, let's review what you've learned, some key takeaways, and what you can do next. The main goal of this course was to expand your Terraform skills through the lens of Microsoft Azure, that included basic things like all the options that exist for authenticating to Azure with Terraform and the different providers that are available for the Azure services. While we mainly focused on the AzureRM provider, we also touched on the AzAPI provider and the

Azure AD provider. Terraform can also leverage Azure for the storage of state data, so we covered how to properly set up an Azure Storage account for that purpose and how to configure Terraform to use it. There's a really good chance you've got existing resources in Azure that you want to manage with Terraform. So we took a look at both the Declarative Import process and you got a chance to take the Azure export for Terraform tool for a spin in the labs. We also got into some of the fundamental building blocks for Azure like virtual networks, virtual machines, and load balancers all through the lens of using modules. Honestly, these essential IAS elements are still the most popular objects folks deploy in Azure. Automating Terraform is a key part of your journey to infrastructure as code mastery, so we covered how to use Azure DevOps to automate the deployment of Terraform configurations. And then finally, we wrapped up with a look at situations where the AzureRM provider is missing some features and how you can leverage the AzAPI provider to fill in the gaps. That was a lot of ground to cover, and you've done a great job working through it all. Beyond the learning objectives, what are some key takeaways from the course? For starters, you may have noticed that we used a lot of modules to deploy resources, VNets, VMs, AKS clusters, etc. Using modules is a great way to enshrine best practices and ensure consistency across your deployments. Microsoft is actively working on producing verified modules for Azure, so I would keep an eye on that project. Another key takeaway is the importance of understanding how Azure services actually work. Terraform provides a common language and process for the deployment and management of infrastructure, but it doesn't replace the need to understand the services you're deploying. Modules can help remove some of the guess work, but you still need to understand what you're deploying and why to select certain options over others. I also want to stress the importance of automating your Terraform deployments. In the course end lab, we used Azure DevOps, which is a great tool, but it's certainly not the only tool out there. GitHub Actions, Terraform Cloud, GitLab, and more all have great integrations with Terraform. Automating your deployments is a key step in reducing human error and ensuring consistency across your environments. The last thing I'll say is that Azure can be weird at times. It's a massive platform with hundreds of different APIs all hiding behind the general AzureRM provider. Inconsistencies will pop up. Eventual consistency can be a real pain. You're definitely going to bump your shins on some sharp corners. Fortunately, there's an amazing community of Azure specific Terraform users out there, and they are always willing to help out and share their experiences. So what's next for you? I have a few recommendations. If you're still learning about Azure, there are a ton of great courses on the platform. AKS is one of the most popular services in Azure, so I'd recommend starting there. It ties a bunch of elements together like virtual machines, networking, and storage. It's a great way to see how all the pieces fit together. If you're ready to step up to the next level of Terraform. Check out my deep dive course. We go into a lot more detail on the Terraform language, how to write modules, and how to use the Terraform Cloud service. If certification is your goal, the Terraform Associate

Certification is a great way to go, and keep an eye out for the forthcoming Terraform Authoring and Operations Professional certification. There will be an Azure specific version of the exam coming sometime in the near future. I hope you've enjoyed this course and learned a lot. I'm jazzed that you took the time out of your busy schedule to learn more about two topics that I'm passionate about, Azure and Terraform. If you'd like to connect with me, you can find me at my website, nedinthecloud.com, or on LinkedIn. I'd love to hear more about your experiences with the course and any feedback you have. Until then, go build something great.