Hands-On with Terraform on Azure

Hands-On with Terraform on Azure

Course Introduction

Hello, Cloud Gurus, and welcome back! In this lesson, we're going to explore some introductory concepts that will help us along the way as we complete this course. Let's take a look at what we're going to cover in this lesson. First up, we'll take an introductory look at Infrastructure as Code and Terraform as an Infrastructure as Code tool. Then we'll take a moment to understand the Terraform workflow. And finally, I'll introduce you to our scenario company that we will be deploying some Azure infrastructure for using Terraform. Let's jump in. All right, let's just spend a brief moment bringing everyone up to speed on why Infrastructure as Code is so awesome. To help us understand Infrastructure as Code, let's use the idea of a recipe, and let's make use of a recipe to make a delicious chocolate cake. If you take a look at a chocolate cake recipe, there are really two main parts, the list of ingredients, and the method to create the cake. You collect all of the ingredients and then you make the cake according to the method. If you don't follow the method exactly, you're going to have some serious problems. And when it comes to deploying Infrastructure as Code, if you use scripting tools like Azure PowerShell and the Azure CLI, you really have to create the full recipe. You need to specify all of the ingredients and in what order those ingredients should be used. And this is referred to as imperative programming. If you use Infrastructure as Code tools, like HashiCorp's Terraform and Azure Bicep, you can skip the method part of the recipe, as that will be handled by either the Infrastructure as Code tool or the platform that you're deploying to. All you need to specify is the ingredients to get what you want. You declare what you want. This is referred to as declarative programming. Terraform, like most Infrastructure as Code tools, is a declarative programming tool. Another advantage of Infrastructure as Code is that you don't need to build any of the conditional logic to handle resources that might already exist. If you use an Infrastructure as Code tool to declare you would like a chocolate cake, once you have one, there's no need to recreate that chocolate cake every time you deploy. And this is referred to as idempotency. It's a crazy word, but it basically means that you can run your configuration over and over again and get the same result every time, which in our case would be a delicious chocolate cake. Now, if you use an imperative programming language, you would need to build in the logic to handle situations where someone had already some of the ingredients in the bowl. Terraform handles this by managing the state of your environment. And Infrastructure as Code tools like Terraform also automatically handle dependencies. Now some do this better than others, but that means that the order of the deployment is managed for you, including deploying resources in parallel when possible. And Infrastructure as Code tools like Terraform also work really well with version control systems like Git, allowing you not only to have a readable version of your configuration, but version that configuration and control how it is released. Terraform integrates with version control systems through Terraform Cloud. And finally, the last major advantage of Infrastructure as Code is reusability. Terraform and other Infrastructure as Code tools allow you to use modules to reduce the amount of code you copy and paste by referencing a module and calling that module over and over again to create lots of similar objects without heaps of additional code. All right, so how does Terraform work? At the dawn of time, we had our cloud provider. In this course, our cloud provider is Microsoft Azure. So how do we use Terraform to deploy to Azure? It all starts with the Terraform executable, which you can download from HashiCorp. It's free and available for heaps of platforms. And you'll use the Terraform executable to perform a range of functions. In this course, we'll use it to validate and deploy our configuration, and we'll also use it to log in to Terraform Cloud. And once you have the Terraform executable file, you can create your configuration files and declare the configuration that you want. And you can have one or more configuration files in the directory. And these files can be written using either HashiCorp configuration language or JSON, but everyone uses HashiCorp configuration language, so that's what we'll be using in this course. And these configuration files include which providers you'd like to use and how you want to use those providers. Providers are plugins that act as the interface between the Terraform executable and your cloud provider. They translate your configuration into actual resources in the cloud provider and which resources you'd like to deploy and which attributes or properties you'd like those resources to have. And you can also reference existing resources through data sources and save time by reusing code with variables and modules in the configuration. And once you have your configuration, you'll initialize it by running Terraform init. At this point in time, Terraform will read your configuration and download any required providers from a registry. Typically this is the Terraform public registry, but you can use any registry. And Terraform will also initialize the backend, which is where the current state of your infrastructure is stored. Now, if you don't specify where to store your state via our backend in your configuration file, Terraform will use a local state file, which might be an existing state file or it might create a new empty state file. And once initialized, you can run terraform plan. Terraform will read your state file and use it to determine which changes, if any, need to be made. And if you're happy with the changes, you can then run terraform apply. And the Terraform executable will use the Terraform provider to deploy your infrastructure to your chosen cloud provider and update the state with the latest changes. Now, in this course, we're going to step through this process locally, but we'll also explore version control, continuous delivery with GitHub Actions, and securely store our state in Terraform Cloud. And we'll step through this entire process, so if any of these terms were unfamiliar to you as we went through them, we'll cover it all in the course. To help us relate to the concepts we're learning in this course, we're going to deploy some Azure infrastructure using Terraform for our fictitious company, River City AI. And we're going to do an excellent job because that's how we roll. Now, River City AI specializes in optimizing speech-to-text for efficiency and accuracy using machine learning. And they're working on next generation speech recognition for generative AI. And River City AI is already using Amazon Web Services, and due to the success and capabilities of Microsoft Data and AI services, they are looking to adopt a multi-cloud environment and drive innovation. Now, River City AI needs to move fast, so agility is important, but so is security. The AI space is moving fast, so we need to strike a balance between agility and security. In this course, we'll deliver agility by enabling continuous delivery through GitHub Actions, reusable code with modules and variables, and allow multiple platform engineers to deploy infrastructure using version control and remote state. And because security is so important, we'll ensure that all secrets are stored securely, including secrets in the state file. We'll also use modules to enable repeatable deployment of secure by default infrastructure. And because River City AI has indicated that it will want to scan the Infrastructure as Code at some point in the future, potentially using something like Terrascan to implement DevSecOps, we'll create an extensible infrastructure delivery pipeline using GitHub Actions. And that's pretty neat, right? All right, let's wrap up with a summary and some key takeaways. First up, we reviewed Infrastructure as Code and learned that Terraform has several advantages, including how its idempotency allows you to achieve consistent results in your deployments by enabling deployments to be run over and over again to achieve the same result. We also took a look at how Terraform modules can be used to create reusable code, and we learned that Terraform will automatically handle your dependencies so you won't need to script out your entire deployment, and how you can integrate Terraform with version control systems like Git to share, collaborate, and version your infrastructure. We also explored the Terraform workflow, including how you can author your configuration files, then initialize your environment to get your state and providers ready, and then plan and deploy your infrastructure all using Terraform. All right, that's all for now. Thanks for being with me. I hope you found this lesson useful, and I look forward to seeing you in the next lesson.

Setting Up Your Infrastructure-as-Code Workstation

Hello, Cloud Gurus, and welcome back! In this lesson, we're going to get your workstation set up for deploying your Infrastructure as Code to Azure using Terraform. Let's take a look at what we're going to cover in this lesson. First up, we'll take a look at the Infrastructure as Code Toolkit we'll be using throughout this course. Then we'll take a look at a demo of how you can automate the process of setting up your workstation using Chocolatey. And now I know this is a short lesson, but I didn't want to leave anyone behind, so let's start from 0, and before we know it, we'll be continuously delivering our Infrastructure as Code. Let's get stuck in. To alter your Infrastructure as Code, you're going to need a few tools in your toolkit. Let's take a quick look at some of the tools you'll need to get this done. Firstly, you'll obviously need the Terraform executable, and we'll use this to validate our configuration, initialize our Terraform providers and state, and plan out and deploy our infrastructure. We'll also need Git, and I have the GitHub octocat icon here, but Git and GitHub are different things, and we'll use Git as our version control system to work with our code in a distributed manner, allowing multiple platform engineers to work with the code. And GitHub will serve as our Git repository. We'll also need the Azure CLI, and we'll use it to log into Azure when we're doing deployments locally. And it's also used to create credentials when deploying our infrastructure using Terraform Cloud. And last, but not least, we'll need an integrated development environment, or IDE, to author our configuration. Visual Studio Code and the Visual Studio Code extensions like the Terraform extension will make authoring our code a seamless experience. All right, let's take a look at a demo of how you can easily set up your Infrastructure as Code workstation. Now because we're all about that Infrastructure as Code life, let's use code to set up our workstation in this demo. I'm going to demonstrate how you can do this on Windows, but I'll also drop some links so you can get the tools set up on your operating system of choice. And I also wanted to show you this because I'll use Chocolatey to set up an Infrastructure as Code workstation for you to complete all of the demos and hands-on labs in this course, and I wanted you to see how I did it, so we're not skipping anything. Now, Chocolatey is a package manager for Windows, and in this demo, we'll use Chocolatey to build an Infrastructure as Code installation script and configure our workstation. And we'll also manually install a couple of Visual Studio Code extensions that we'll use throughout the course. All right, so here I am on the workstation that we're going to use as our Infrastructure as Code development workstation. And I've gone ahead and navigated to chocolatey.org, and we're going to use Chocolatey to build a script that we can use to automate the installation of all the software we need for our Infrastructure as Code workstation. We'll go ahead and click Get started, and then we'll just scroll down a little bit, and we'll copy the installation script, then we'll open an administrative PowerShell, and then we'll go ahead and paste the script in. Now, this script is simply bypassing the execution policy for this particular process, and it's ensuring we're using TLS 1.2, and then it's using the invoke expression PowerShell cmdlet, which is the iex alias there to download the Chocolatey installer script and run that script. Let's go ahead and run that. This will just take a moment to install, so we'll come back when this is done. All the installation is now complete. Now, we just need to close PowerShell and reopen it to refresh our environment variables. So I'll go ahead and close PowerShell, and then we'll open PowerShell back up again. All right, now let's go back to Microsoft Edge and we'll build our Infrastructure as Code installation script. We'll scroll back to the top and we'll navigate home. We'll click Find packages. This is just indicating that we're using a public repository or a community-powered repository, and that could be a security issue, but we understand the risk, so we'll go ahead and click I understand. Now, let's go ahead and search for Terraform, and we'll click the button to add to script builder. And now we'll repeat this process for Visual Studio Code. We'll also grab Git for version control, and we'll also grab the Azure CLI. Now we can go ahead and click the builder icon, and then click Generate Script. We'll go ahead and click Next. We'll ensure individual is selected, and then we'll click Next. We'll go ahead and copy this script, jump over to PowerShell, and paste the script to install the required software. This will just take a moment to complete, so we'll come back when this is done. All right, the installation is now complete. Now, you can see that we'll need to close and reopen PowerShell to refresh our environmental variables, or we can run the refresh env command. I'm just going to go ahead and close PowerShell, and then I'll reopen it. Now, let's confirm the installation of our software. We'll run terraform version. That confirms that Terraform is installed successfully. And then we'll run git version. That confirms that Git is installed successfully. And then we'll go ahead and run az version, and that confirms that the Azure CLI is installed successfully. And then we'll go ahead and run code version, and that confirms that Visual Studio Code is installed successfully. All right, now let's install the HashiCorp Terraform Visual Studio Code extension. To do that, we'll run code with the install extension parameter, and we'll provide the name hashicorp.terraform. All right, that installation is now complete. So we'll go ahead and close all of these Windows and we'll open Visual Studio Code to confirm the extension is installed successfully. And we'll just navigate to the extensions blade, and we can see that HashiCorp's Terraform extension is installed successfully in Visual Studio Code. All right, Infrastructure as Code workstation is ready to go. I'll drop links for how you can do this on Linux and macOS, but we're ready to go with the hands‑on lessons and labs for the remainder of this course. All right, let's wrap up with a quick summary and some key takeaways. First up, we explored the toolkit you'll need to set up your Infrastructure as Code workstation when working with Terraform and Azure, including the Terraform executable, Git for version control, the Azure CLI to log into Azure and generate credentials, and Visual Studio Code with the Terraform extension to author our Terraform configuration. We also took a look at how you can automate the installation process using Chocolatey. All right, that's all for now. Thanks for being with me. I hope you found this lesson useful, and I look forward to seeing you in the next lesson.

Deploying to Azure Using Terraform

Hello, Cloud Gurus, and welcome back! In this lesson, we're going to deploy to Azure using Terraform and its command‑line interface. Let's take a look at what we're going to cover. First up, we'll explore the Terraform configuration building blocks that you will use to declare your configuration. Then we'll take a look at a decent end‑to‑end demo of authoring a configuration and deploying that configuration to Azure using the Terraform command‑line interface. Let's get started. Let's explore the Terraform configuration in more detail by taking a look at the building blocks we will use to define our Infrastructure as Code. The blocks are defined using opening and closing curly brackets, and the attributes or properties that you set for those resources within a block are called arguments. The first block you'll want to consider, and it's essential, is the Terraform block. And this block is used to configure Terraform, including things like the Terraform version you'd like to use, and in a production environment, it's common to bind Terraform to a specific major version to help ensure reliability. It'll also define the backend or Terraform cloud configuration block, and the backend defines where your state is stored, or you can store your state in Terraform Cloud. The other super important setting you want to include in your Terraform settings is the required providers and the version of those providers. We'll be using the azurerm provider throughout this course, as it's the HashiCorp official provider for Azure. The next block you'll want to include in your configuration is the configuration of your providers. We'll only be using the azurerm provider in this course, and you can configure arguments for your providers here. The only argument we'll be using in this course is the skip_provider_registration property, and this property is used to stop Terraform from registering all of the Azure resource providers in Azure Resource Manager. Azure resource providers, not to be confused with Terraform providers, are used to configure which resources are available in your Azure subscriptions. And because we'll be working with Cloud Playground, we have restricted permissions so we can't register all of the Azure resource providers in our playground environment. And you can also see we have an empty features block. This is required, even if it is empty. The next block of code we'll be using frequently in this course is the resource block, and this is where you declare resources that you want to exist and what properties or arguments you want them to have. In this example, we're declaring a resource group and specifying a name and location arguments. One challenge you'll face both in our hands on lab environment and in the real world is dealing with existing resources, like the resource group that has been provisioned for you in Cloud Playground. And we'll explore importing existing resources in the demo in just a moment. All right, let's take a look at an end‑to‑end demo of deploying to Azure using Terraform and its command‑line interface. In this demo, we'll use our Infrastructure as Code workstation that we built in the previous lesson, and we will author our configuration by specifying the Terraform configuration block, the provider configuration block, and two resources, a resource group and a storage account. We'll also initialize the working directory with terraform init, which will download the required azurerm provider and initialize an empty local state file. And because the resource group already exists, we'll import the resource group into our state using the terraform import command. Then we will define a storage account resource format and validate our configuration. Then we'll create an execution plan with terraform plan. And finally, we can deploy our configuration with terraform apply. Let's take a look at the demo. All right, so here we are on our Infrastructure as Code workstation that we set up in the previous lesson. And I've gone ahead and opened Visual Studio Code so we can get started creating our Terraform configuration. We'll go ahead and open a folder, and then we'll create a new folder for our configuration. We'll put it in the documents folder, and we'll create a new folder called Terraform. We'll go ahead and select yes that we trust the authors, and we'll go ahead and close the welcome dialog. Now, we're going to create a new Terraform configuration file. We're going to use the name main.tf. And there are naming conventions for other Terraform files, but we're just going to get started with one file and name it main.tf in this lesson. Let's get started authoring our Terraform file. Firstly, we'll create the Terraform block, and I've just pressed tab to complete that block. Now, we need to set the required version. I'm going to set the required version to at least 1.3.7. And now we'll set the required providers to azurerm, so I'm going to use IntelliSense again with Ctrl+Space, select required providers, and then I'll create a block for our provider requirements. I'll just put an extra space in there. I'll use IntelliSense again, and we'll set the source to HashiCorp with a forward slash and then azurerm. I'm just going to indent this code a little bit here, and we'll set the version to be at least 3.43. Now, I'm just going to put an extra character in here to let Terraform know that it can increment the last digit in this version, and that's okay with me. To do that, I'm going to use the tilde character and then the greater than character. Now, before we continue, if we want better IntelliSense, when we configure the provider and resources for that provider, we should initialize Terraform because it'll go ahead and download that provider and provide better IntelliSense when we author the configuration. So we'll go ahead and open a terminal, and we'll go ahead and run terraform init. But before I do that, importantly, I need to save the file. Now it's gone ahead and downloaded the Terraform provider for Azure, and let's go ahead and create our provider configuration block. We'll go down to the next line, and I'll just press tab to complete that. IntelliSense is prompting azurerm, and that's the provider we'd like to declare. Now, we need our empty features block as we discussed in the lesson. And because we're working with an environment with limited permissions, we need to provide the skip provider, registration argument and set the value to true. I'll just use control space to bring up IntelliSense, start typing skip_provider_registration and set that to true. All right, now we can start to declare our resources, and because I'm using Cloud Playground, I have an existing resource group, so let's declare and import that resource. First, we'll create a resource block for the resource group. So I'll start searching for the name. I'll type azurerm_, and then start typing resource group. And we can see that's one of the options here, so we'll go ahead and select that. We'll just provide the local name, rg. And now let's use IntelliSense to determine the required properties. I'll press Ctrl+Space, and we can see that location is required. A name is required. So let's go ahead and select the name argument. We'll leave this value blank for now, and we'll add the other required properties with IntelliSense again, which is the location. I'm going to leave that blank for now as well. Now because this is an existing resource, let's go ahead and get those properties from the Azure portal. We'll open up Edge, and then we'll view the properties for this resource group. We'll go ahead and copy the name, and we'll paste that into our code, and then we'll go ahead and copy the location, and we'll paste that into our code as well. Now because this is an existing resource, we need to import it into our Terraform state file. To do that, we'll use the terraform import command. But before we do that, Terraform is going to need some access to Azure, so let's use the Azure CLI to log in to Azure. To do that, we'll use the az login command. We're successfully logged in, so we can go ahead and close this browser tab and navigate back to Visual Studio Code. Now, let's import that resource group using the terraform import command. We need to provide the type of resource we'd like to import and the local name for that resource, which is rg in our case. And then we need to provide the resource ID, and we're going to grab that from the Azure portal. I'll copy the resource ID, jump back to Visual Studio Code, and paste that in. We'll go ahead and run that command. But before we do that, let's save our Terraform file, and then we can go ahead and run the command. All right, it's initializing our Terraform state and importing that resource group into our state file. All right, with our existing resource group imported, then let's go ahead and define another resource. And in this case, we'll create an Azure storage account. We'll just go down to the next line. We'll start declaring a resource B block. We'll start topping the name for the Azure resource, which is azurerm, an underscore, and then storage, and we'll select storage account. I'll just provide a name for this resource. I'll just call it storage, which is quite creative. And then we can start to go ahead and configure the arguments or attributes that this resource will have. Just go down to the next line, and we'll tab in a little bit here. And we'll use IntelliSense again. We can see that the access tier is optional. The account kind is optional. The account replication type is required. So we will set that, and the account tier is required. Are there any other required values here? Just go down through these. The location is required, which we'd expect, and the name is required. All right, so let's go ahead and start with the name, and I'll just provide a globally unique name for our storage account. And we know that we need to set the location as well, so we'll go ahead and define that argument. Now because I want this storage account to use the same location as our resource group, I'm going to provide a reference here to the resource group's location to save me typing out Central US for the location for the storage account. Now, we need to define the other required properties. We'll go ahead and use IntelliSense again, and we need to have the account tier and the account application type. So we'll start with the account tier, and you can see here IntelliSense is not really providing us with any useful information. So where can we look for the values that account tier is expecting to have for this argument? Well, for that, we can use the documentation in the Terraform registry. So we'll open Microsoft Edge. We'll go to the Terraform registry, which is registry.terraform.io, and we'll browse providers. We'll select the Azure provider. And then we'll go to the documentation for this provider. Then let's start searching for the resource type that we're defining, which is an azurerm storage account. And we'll select the storage account resource, and then we'll go down to the argument reference, and we're looking to set the value for account tier. And we can see here that it accepts either standard or premium, so we'll go ahead and use standard. Now, let's go ahead and add the other required value, which is the account replication type. And again, IntelliSense is not giving us any useful information here, so we'll go ahead and grab that from the registry. We'll go ahead and use, read-only geographic redundant storage, so we'll copy that and paste that into our configuration. All right, with our resource declared, let's format and validate our configuration so we can format the document by right-clicking the code and selecting Format Document. Now, let's save our file, and we'll run terraform validate to ensure our configuration file is valid. All right, and we can see here we're missing a resource group argument, so we've just missed that required value. Let's go ahead and provide our resource group argument for our storage account and its resource group name. And then we'll use our existing resource group, and we'll use its name property. Let's go ahead and format our document again. We'll save our document. I'm just pressing Ctrl+S to save the document, and then we'll run terraform validate again. All right, our configuration is valid. Now, let's go ahead and plan our execution by running terraform plan. Now, we'll actually save our Terraform execution plan out to a file. This is completely optional, but we can use this execution plan when we run terraform apply. So let's go ahead and run through that process. We'll run terraform plan, we'll use the -out parameter, and we'll name it tfplan. We'll go ahead and run that. And we can see that it's going to add one resource, and it's saved the plan to the tfplan file. Let's go ahead and deploy our configuration using the terraform apply command and passing in our plan file. Now because we're using a saved terraform plan file, terraform apply doesn't run the plan command again. Instead, it just accepts the plan that we've provided. This will just take a moment to create, so we'll come back when this is done. All right, the apply is complete. Let's go ahead and take a look at our new storage account in the Azure portal. Navigate back to the resource group, and we can see we have our new storage account configured with the properties that we set in our configuration file. That wraps up this demo. All right, let's wrap up with a quick summary and some key takeaways from this lesson. First up, we explored the building blocks that make up your Terraform configuration. Common blocks include the Terraform configuration block where you configure Terraform settings like backend configuration and the required providers. And the provider block, which is used to configure providers, and resource blocks, which are used to declare your resources. Now, those aren't the only blocks you'll define in your configuration, and we'll take a look at a couple more later on in the course. And while we only worked with a local state file in this lesson, we did learn that we can use either a backend configuration or Terraform Cloud configuration in the Terraform block to use remote state. And remote state is something we'll explore in more detail in the following lessons. We also took a look at an end-to-end demo of how you can deploy to Azure using Terraform, including, importantly, how you can import existing resources into your state file using the terraform import command. All right, and now it's your turn. There's a hands-on lab immediately after this lesson where you can try the steps that we've completed in this demo.

Good luck with the hands-on lab, and I'll see you in the next lesson!

Managing Terraform State

Hello, Cloud Gurus, and welcome back! In this lesson, we're going to take a look at how you can manage your Terraform state. Let's take a look at what we're going to cover in this lesson. First up, we'll take a moment to explore some of the good things about Terraform state. Then we'll take a moment to understand some of the downsides and how we can mitigate those. Then we'll finish up by stepping through a demo where we will migrate our state to Terraform Cloud. I can't wait! Let's get started. In the previous lessons, we mentioned idempotency, and that means that Terraform can apply the same configuration over and over again, and you'll get the same result. By default, when you run terraform plan, Terraform will create a local state file. And the state file is in JSON format, or JavaScript Object Notation format, and it has the name terraform.tfstate, and it's placed in the same working directory that contains your configuration. You can think of Terraform state as a dictionary that maps what you've declared in your configuration with what has been deployed in the real world. And it's this mapping and the associated metadata in the state that allows Terraform to determine and track dependencies automatically. But having a local simple state file also has another advantage, and that is around performance. If Terraform didn't keep an efficient map of your resources, it would need to query the provider's API every time a deployment is run to determine the current state of a resource before it attempts to deploy or update that resource. Not doing so could cause Terraform to try and deploy something that already exists, which would result in an error. And reading a simple state file is really fast. By default, Terraform will automatically refresh your state when you run terraform plan, which works well for small deployments. But you don't need to do that. If you have a large number of resources, you can skip the refresh step and save deployment time significantly. Now, we've explored why Terraform state is great, but what are some of the challenges when working with Terraform state? Well, the first and probably the most important challenge is that Terraform state can include sensitive data, and this sensitive data might include things like default passwords and connection strings. And because state is stored in a plain text JSON file locally by default, this means you can have sensitive data stored in plain text on your PC. Now we all know this is a huge no-no. But you can get around this limitation by using a backend. We mentioned backends briefly in the previous lessons. A backend allows you to store your state remotely, that is, not on your PC. And there are a number of different backends supported by Terraform by default, including Azure Blob Storage. Now, I don't know about you, but I am not interested in storing sensitive data in an Azure storage account unless I absolutely have to. So we're going to take a look at another alternative, and that is storing your state in Terraform Cloud. If you store your state in Terraform Cloud, your state will always remain encrypted at rest and in transit without you having to manage all of that security. But there is another advantage to remote state. If we have a team of platform engineers working on our configuration, it's not going to be much use to them if it's stored on my computer and I go on a lovely vacation. Remote state enables collaboration between multiple engineers working on the same configuration at the same time. Terraform will automatically lock and unlock the state when required. Now, I know I said this was bad news, but really it's just more good news if you take the time to implement these state best practices. Now, remote state is essential for us in meeting the requirements that River City AI had back in lesson one. They indicated that remote state helps them meet their agility goals, and it's just a bonus that it's also more secure. All right, let's take a look at a demo. In this demo, we'll take a look at how you can migrate your state to Terraform Cloud. We'll start by creating and viewing a state file, then we will create a Terraform Cloud workspace, and we will upgrade our configuration to migrate the state to Terraform Cloud, and finally, we'll go ahead and delete that local state file. Let's take a look at the demo. All right, so here I am on the desktop of our Infrastructure as Code workstation, and I've gone ahead and logged into Azure using the az login command, and I've brought in an existing configuration that we can use to create local state and then migrate that local state to Terraform Cloud. You can see here that I have the Terraform configuration, I have the provider configuration, and I have one resource group configured. Now, this resource group already exists in Azure, but we haven't imported it into Terraform state, and you can see that we don't have a Terraform state file created locally. Let's go ahead and run terraform init to initialize the working directory and download the azurerm provider. With our working directory initialized, now, let's import that resource group into our Terraform state. We'll do that using the terraform import command, and we'll provide the reference to the resource that we want to import, which is an azurerm_resource_group with the name rg. And then we need to provide the ID for that resource. So I'll just jump over to the Azure portal and grab that resource ID. Now, we can go ahead and run that import command that will create a local state file and import that resource into our state file. We can see the state file has been created locally, and we can use the terraform show command to view our Terraform state. All right, let's move this date to Terraform Cloud. I'll jump over to the browser, and we'll navigate to Terraform Cloud. And you can see the URL for Terraform Cloud is app.terraform.io, and I've gone ahead and just created a free Terraform Cloud account, and I've gone ahead and created an organization as part of the Getting Started wizard. Now let's go ahead and create a workspace to store our Terraform state. We'll select Create workspace. We're going to run Terraform from the Terraform CLI, so we'll select CLI‑driven workflow. But if we wanted to run it from something like GitHub, we could use version control, or if we wanted to use something external to control Terraform Cloud, we could use an API‑driven workflow. And that's something we'll look at in future lessons in this course. But for now, we'll select CLI‑driven workflow. I'm just going to call this workspace remotestate, then I'll go ahead and click Create workspace. Now, Terraform Cloud provides the cloud configuration block that

we'll need to include in our Terraform configuration in our main.tf file. So I'm going to go ahead and copy this cloud code block and paste that into Visual Studio Code. And I'm going to paste this within the Terraform block. I'll just format the document and I'll take out that extra line on line 16 and that extra line on 11. We'll go ahead and save the configuration now. Now we'll run terraform login to log into Terraform Cloud. Now, Terraform is just prompting that it will navigate to app.terraform.io and request an API token, so we'll type in yes to confirm. I'll use the default description for the API token and select Create API token. And then we'll copy this API token, and we'll paste that into the terminal and Visual Studio Code. Then we can go ahead and present her. All right, we've successfully logged into Terraform Cloud. Now, I just want to call out why the API token has some advantages over local state and why it potentially has some issues. So the first is that the API token is now stored in my user profile, it's not stored in my working directory, so it's less likely to get checked into source control and then potentially shared with people it shouldn't be shared with. It's also short-lived, and I can easily delete those API tokens if I no longer need them. But for the best possible security for your API tokens, you should probably consider using a credential helper, and I'll link to the resources for this lesson for some details on how you can set up a credential helper, and I'll provide a link to Terra creds, which is an excellent credential helper that you can use with Terraform to protect your API tokens. But for now, we'll continue on with a local copy of our API token. Let's go ahead and run terraform init again, and this will initialize Terraform
Cloud. Terraform is now asking if we should migrate our existing state to Terraform Cloud. We definitely want to do that, so we'll go ahead and type yes. All right, our state has now been migrated to Terraform Cloud. Let's jump back over to Terraform Cloud and have a look at our remote state. We'll navigate back home. We'll navigate into our remote state workspace, and then we'll select states. And we can see that our state has been migrated to Terraform Cloud. Now, let's go ahead and delete our local state file. We'll jump back to Visual Studio Code. To do that, we'll run rm terraform.tfstate and then provide a wildcard to delete both the Terraform state and the Terraform state backup. And we can see that our local Terraform state files have now been deleted. All right, that concludes this demo. All right, let's wrap up with a quick summary and some key takeaways. First up, we explored Terraform state and how it is used
for configuration mapping and dependency mapping. We also took a look at how state can improve performance when working with large configurations to reduce the number of requests that need to be made to a cloud provider's API. We also explored how you can protect sensitive data in Terraform state by migrating your state to Terraform Cloud. And finally, we explored how remote state enables collaboration on your Terraform configuration. Now, there is a hands-on lab up up next that you can try to step through this process yourself, and I encourage you to give it a go. But that's all for me from now. Thanks for being with me. I hope you've found this useful, and I look forward to seeing you in the next lesson.

Providing Continuous Delivery

Hello, Cloud Gurus, and welcome back! In this lesson, we're going to explore how you can provide continuous delivery of your Infrastructure as Code. Let's take a look at what we're going to cover in this lesson. First up, we'll very briefly review the basics of version control. Then we'll take a moment to explore the basics of continuous delivery. And then we'll put it all together and explore an end-to-end continuous delivery workflow. And then we'll finish up with a demo of providing continuous delivery of Terraform Infrastructure as Code with GitHub Actions. Let's jump in. So at this point, we have remote state which is great because it's securely stored in our Terraform cloud account, and that state is accessible by other platform engineers. But we do have another problem to solve. Our Terraform configuration files are still stored on our local computer, and we need a way to centrally store this configuration so other engineers can view and work on the files. This is where a version control system comes in to save the day. Version control systems allow you to store your code centrally in a repository and provide access to other engineers as required. And because code is versioned in the repository, you can review the changes made over time. And you can also collaborate on that code. You can review each other's changes and use branches to protect production workloads from unintended changes. Now, Git is the industry standard version control system, so we'll focus our efforts on Git in this course, and GitHub is one of many web-based implementations of Git. And GitHub, like most web-based version control systems, not only provides a repository for our code, but also some other valuable additions. Let's take a look at those now. Having all of our code in a central location is great, but we'd still have to do all of the deployments manually from the command line by running terraform plan and terraform apply, which might be okay, but I don't know about you, but I want to be able to go on vacation. So it would be nice if I can automate the process of deploying the infrastructure whenever changes are made to the configuration of that infrastructure. And that's what continuous delivery is all about. Based on events that occur in our code repository, we can trigger an automated workflow that will update our infrastructure. Now, this can be done easily using Terraform's Version Control System integration feature, but that's where Terraform Cloud functionality is a little limited. It's only capable of automating the deployment of our Terraform infrastructure. What if we wanted to add more tasks to our workflow, like testing the code to make sure it meets company or industry standards? Or what if we wanted to also automate the deployment of the application that runs on the infrastructure as part of the same workload? Well, that's where GitHub Actions comes in. GitHub Actions is a continuous integration and continuous delivery platform, more commonly referred to as CI/CD. With GitHub Actions, you can automate your workflows like testing and deployment. Continuous integration is about testing and building applications. Continuous delivery is about deploying those applications to an environment like your production environment. The automation that powers

CI/CD is often referred to as a pipeline. In GitHub Actions, workflows are completely customizable, which meets the requirements that River City AI indicated way back in lesson one. Let's take a look at an end‑to‑end workflow that enables continuous delivery of our Terraform Infrastructure as Code. It all starts with a repository for our code, and we'll use GitHub to host our repository. And that repository will have branches that represent our environments and work in progress. In our repository, we'll have a main branch that represents our production infrastructure, and in our environment, that main branch will be protected so changes can't be made to that branch without an appropriate approval taking place. To make changes to our production infrastructure, we'll clone that repository, which copies that code to our workstation. Then we will create a new branch for the changes we want to make to the infrastructure. And this is commonly referred to in the industry as a feature branch. And then we'll make some changes to that infrastructure, stage and commit those changes using our version control system, Git, and push those changes to our GitHub repository. And then we'll make a request to move those changes that we've made to production. And this is referred to as merging your feature branch into the main branch. To merge your code from one branch to another, you create what's known as a pull request. At this point, GitHub Actions will run a workflow to test the code if required, and if required, someone else can be asked to review the changes. And these workflows are completely customizable based on your requirements. And once we're happy with the code, the pull request can be completed and another workflow, or the same workflow with workflow conditions, can run that will deploy the infrastructure to production. All right, let's take a look at a demo of configuring continuous delivery for our Terraform infrastructure using GitHub Actions. In this demo, we will create a GitHub repository, and then we will configure GitHub Actions to provide continuous delivery, and we'll configure branch protection on the main branch to protect our production workloads from unauthorized changes. We'll also briefly configure Terraform Cloud, and then we'll go through the Git workflow to clone, author, stage, commit, and push our configuration. And then we will approve our changes and watch the GitHub Actions workflow deploy our infrastructure. Let's take a look. You can see I've signed into GitHub here. Let's go ahead and create a new repository for our code. Select New. Looks like that I'll be the owner of the repository, and then I'll provide a repository name. I'm just going to use the name TerraformCI. I'm going to create a public repository. I'm going to add a README file. And I'm going to add a Terraform .gitignore. We'll go ahead and hit Create repository. Now, let's configure continuous delivery. We'll select Actions. We'll search for the Terraform workflow. We'll go ahead and hit Configure. Now, there's some comments at the top of this file. We can go ahead and skip past all of the comments. It's just explaining how to use this workflow. The on property defines when we'd like the workflow to run. We want our workflow to run only when we push changes to the main branch. So it's going to run on, push on the main branch. We can go ahead and delete the pull request option, and then it's got a list of the jobs that this workflow will run. I'll just scroll down a little bit. And this workflow isn't doing anything special. It's just taking a copy of our code with git checkout, and then it runs through the Terraform workflow that we've been doing manually up until this point, including terraform init, format, plan, and apply. And you can see there's an if statement here that we don't need, so we're going to remove that if statement. Now we can go ahead and commit this workflow file to the GitHub workflows directory. So we'll hit Start commit. Now you can see we're committing directly to the main branch here, but we don't have any Terraform files yet, so this will fail. That's okay for the purposes of this demo. We'll go ahead and click Commit new file. Now, let's add the secret that this workflow requires to access Terraform Cloud. We'll navigate to Settings. Then on Secrets and variables, we'll select Actions, and then we'll create a new repository secret. We're going to provide the name that was in our workflow, which is TF_API_TOKEN, and now we need to get a token from our Terraform Cloud account. So we'll navigate over to Terraform Cloud. We'll click our user account. Click User Settings. Go to Tokens. Request a new API token. I'll provide the name GitHub Actions, and then create the API token. We'll go ahead and copy this, and then we can click Done. We'll jump back over to our secret and we'll paste in in the API token. Click Add secret. So we now have our repository and our GitHub Actions workflow and an API token that GitHub Actions can use to access Terraform Cloud. Now, let's go ahead and configure some branch protection policies to protect our main production branch from unexpected or unauthorized changes. We'll select Branches under Code and automation. We'll click to add a branch protection rule. We'll set the pattern to main, so we're protecting our main branch. We'll select require a pull request before merging and set it to require approvals for at least one person. And we'll also require status checks to pass before merging. This would normally be used for testing. We don't have any tests yet, but River City AI indicated they might want to do this in future, so we'll go ahead and enable this check box. And we'll select the option to not allow bypassing the above settings. And then we'll hit Create. All right, our branch protection rule is in place. Now, let's go ahead and configure Terraform Cloud. We'll create a new workspace, navigate home, create a new workspace. We're going to use GitHub Actions to power the workflow, so that'll be an API‑driven workflow. I'm going to call the workspace name TerraformCI, and then we'll create the workspace. Now, let's go ahead and create some environmental variables that Terraform Cloud will use to access Azure. We'll select Variables. And then we'll click to add a variable. We need to create four environmental variables, so I'll go ahead and create those, and then we'll come back when we're done. Okay, I've gone ahead and added in all of the required variables, including the subscription ID, the client ID, and tenant ID, and the client secret, and you can see that I've marked the ARM client secret as a sensitive variable. Now, let's go ahead and author our configuration. We'll go back to GitHub. We'll navigate to code. We'll select Code, and we'll copy the repository URL. Then we'll open Visual Studio Code. I'll press Ctrl+Shift+P as displayed here to bring up the Command Palette, and I'll type git clone. I'll paste in the repository URL that I just copied. I'm going to clone this repository into my documents directory, and then click select as repository destination. We'll go ahead and open the repository. And we'll select to trust the authors. The next thing we need to do is configure the Git client using the git config commands to set our username and our email address. So I'll just bring up a terminal, and I'll run two Git commands to configure Git with our username and email address. And I'll just paste those in so you don't have to wait for me to type them. All right, I'll clear the screen. Now let's create a new feature branch to author our infrastructure. We'll select the branch icon in Visual Studio Code. We'll select to create a new branch. We'll call this add infrastructure. And we've now changed to that branch, as you can see in the bottom left. Now let's author the Terraform code. I'm going to create a main.tf file, but I won't make you wait while I author that Terraform code. I'll come back once it's authored and we're ready to go. All right, with our Terraform code authored, including our cloud configuration, we can now format and save the configuration. Now, we'll select the source control option in our sidebar. We'll stage the two changes, which includes our dependencies and our main.tf file. Now we can go ahead and commit our changes. I'll just provide a description and then click Commit. Now, we can go ahead and publish our branch. We'll just need to sign in with GitHub, so I'll go ahead and sign in. And now our branch has been pushed to our remote repository, which is on GitHub. Now, let's go ahead and create and complete a pull request. We'll open our browser and we'll go to GitHub. We can now go ahead and create a pull request. So I'll select Compare and pull request. And then we'll go ahead and create a pull request. And we can see that merging is blocked because a review is required. So we could go ahead and add a reviewer if we wanted to. I'm just going to remove the setting that we added earlier in the demo that prevented us from bypassing these settings, and then I'll come back when that's done. And then we can go ahead and merge the pull request without our reviewers. Now, normally you wouldn't do this in a production environment, I'm just doing it so we can complete the demo. And we'll select to merge the pull request, and then hit Confirm merge. Now we can navigate over to GitHub Actions. And we can see how GitHub Actions workflow is running. We'll select that. We'll select the job. All right, that workflow is now running, so we'll come back when that's done. All right, the workflow is now complete. Let's navigate over to the Azure portal and view the results. We'll just go ahead and refresh this. And we can see our new storage account has been created with the properties that we set in our Terraform configuration file. All right, let's wrap up with a quick summary and some key takeaways from this lesson. First up, we explored version control and learned that a version control system allows you to centrally store, version, and have your code reviewed by others. And we learned that branches in the Git version control system are used to represent environments or work in progress, which is commonly referred to as a feature branch. And we also learned that a pull request is used to merge code between branches and that you can protect branches and use pull requests as a gatekeeper between those branches. All right, now it's your turn. I have a hands-on lab after this lesson that you can try to complete these steps on your own. But that's all for now. Thanks for being with me. I hope you found this lesson useful, and I look forward to seeing you in the next lesson.

Effectively Managing Terraform Code

Hello, Cloud Gurus, and welcome back. In this lesson, we're going to explore how you can effectively manage your Terraform code. Let's take a look at what we're going to cover in this lesson. First up, we'll explore how we can leverage Terraform variables to make our code more reusable. Then we'll take a moment to learn about Terraform modules. And then we'll finish up with a demo where we will create a Terraform module and publish it to a private registry. Let's jump in. Up until now, we've explored fairly basic Terraform configurations, and we've been hardcoding all of the required values into that configuration. But there is a problem with this approach. If platform engineers all author their own Terraform configurations and deploy those resources to Azure, they're going to end up using whatever arguments and names and tags they want to when they deploy their resources. And this will lead to some very inconsistent resources. It's a best practice to make sure resources are tagged and named appropriately. Consistent tagging and naming of resources helps with determining the purpose of those resources and it also helps with cost management. So how can Terraform help us achieve resource consistency? Well, the first way is to use variables. When we declare our Terraform configuration, Terraform supports a few different variable types, including input variables, which allow you to pass values to a configuration prior to the plan phase to customize that configuration. For example, you might have a Terraform configuration like this that deploys a storage account, and you can pass the replication type as an input variable, and when you deploy development or test versions of these workloads, you could use a replication type that has less redundancy than the production workload. And as you can see here, I'm defining an input variable with the name storage_account_replication_type with a default value of locally redundant storage. And that can be changed to globally redundant storage when deploying a production workload. And you can see here we're referencing that input variable with the var keyword. Another way to standardize your configuration is by using locals. Sometimes referred to as local values, locals allow you to set a value in your configuration and use it multiple times without repeating the initial expression that defined that value. As you can see in this example, you might set a local value that defines a naming convention and includes the workload name, environment name, region, and instance number. And then you can use locals throughout your configuration for all the resources of a given workload using the local keyword. You can even combine input variables and locals for even more power. And the final variable type to be aware of is output variables. Outputs allow you to essentially export data from one Terraform configuration to another. You can see the output variable defined here is passing the primary blob host for the storage account as an output named storage_account_primary_blob_host, which is the property of the storage account once that storage account has been deployed. Now, let's explore another way to improve the consistency and reusability of your Terraform configurations by exploring modules. Each and every Terraform configuration you define in a directory, be it one Terraform file or a collection of Terraform files, is actually a Terraform module. And modules are another way to reduce rework and improve consistency. Now, you can deploy a module directly as resources using the provider, and that's what we've been doing so far in this course. But it doesn't stop there. You can use Terraform modules to provide a reusable set of configuration that includes a resource or set of multiple resources that are frequently deployed together. Then you can either use the module locally, as you can see. In this example, there is a terra form module name storageaccount that is being used from a subdirectory in that same Terraform configuration working directory. Or you can make it available to more people by uploading the module to a registry. You can use a public registry like the Terraform public registry, or you can create a private registry to share that module internally. You can see the configuration here is referencing a Terraform public registry module. And as you would expect, you can combine the variable types we just explored with modules for maximum reusability. This is a powerful combination that improves resource consistency and speeds up deployment of commonly deployed resources and reduces the times engineers will need to copy and paste code. All right, let's put all of this goodness together and take a look at a demo of creating a private Terraform module registry. In this demo, we will create a GitHub repository to store our Terraform module. Then we will author our reusable Terraform module that creates a secure storage account so that our scenario company, River City AI, can use this module to deploy multiple storage accounts for training different AI models. We will then use Terraform Cloud to create a private registry for our module. And finally, we will create a Terraform configuration and use the module we've created to deploy a secure storage account. Let's take a look. All right, so here we are back on our Infrastructure as Code workstation that has all the tools we need to author Terraform configurations. I'm in my GitHub account here, and we're going to go ahead and create a repository that we can use to host our Terraform module. We'll go ahead and create a new repository. Now, we need to select an owner. So I'll just select myself as the owner. And then we need to provide a repository name, and the repository name needs to be in a very specific format. It needs to use the name Terraform, a dash, azurerm, which is the provider that this module uses, and then the name for the module. We can include a description if we'd like. I'm going to make it a public repository. I'm going to add a README file because the README file will be included in Terraform Cloud to make it easier for people to consume my module. I'll add a Terraform .gitignore, and then we'll go ahead and create the repository. Now, let's author our Terraform module. To do that, we'll clone this repository down locally using Visual Studio Code and then create the configuration for our module. Now, we've been through the process of cloning a repository, o I'll clone this repository locally and then we'll come back when that's done. Now, it's a best practice to have at least three files for your Terraform modules. They are our main.tf file, a variables.tf file and an outputs.tf file. So I'll go ahead and create each of those files. And now we can go ahead and author our module. I'm going to open the main.tf on the left and variables.tf on the right. Firstly, we need to create our Terraform block. We've been through this a few times now, so I'm just going to go ahead and paste that in. Now, I'm going to save that, and we'll use the terminal to run terraform init to download the required provider. Now, we don't need to add the provider configuration block when we're creating a module. The provider configuration will be defined in the parent module. So we can skip over that. But let's go ahead now and create a local variable that we can use to assign tags to resources that are created by the module. And this could be easily extended to be applied to multiple resources in the module. We'll go ahead and create a locals block, add a tags local, and then we'll create a map, which is just a map of key‑value pairs, and we'll set the environment variable to the variable environment. Now, we're using a variable here that we haven't defined yet, but we'll go ahead and define that variable in variables.tf in just a moment. Now let's go ahead and create the resource block for our storage account. I'll just Enter down a few times, and I'll paste that code in just so you don't have to wait for me to type, and then i'll go ahead and explain the code. All right, in this code block, you can see I'm using input variables to define the resource group name, location, and storage account name. I'm hard‑coding some values that I don't want module consumers to change, including the account tier and the public network access. And I'm using a conditional operator to automatically set the account replication based on the environment, GRS for production; otherwise, we'll use LRS. I'm also setting the tags to the local variable name tags. Now, we need to create the variables that we're using in this module. So in variables.tf, we'll create a variable, and I'll just use IntelliSense to create an empty variable. The name of the first variable we're going to create will be the resource group name, which we're using on line 19 in main.tf. It's going to be a string, and we're just going to provide a description of the resource group name. All right, now I'll go ahead and create the remaining variables, and I'll come back when that's done. Now, let's go ahead and define an output variable. It's a best practice to define as many output variables as you can, but we're just going to define one in this demo, so I'll open outputs.tf, and we'll create one output variable, and we'll just use IntelliSense for the code snippet. I'm going to output the storage account ID, and I'm going to set the value to the ID for the storage account that we're deploying in main.tf. And then I'm going to set a description to just be the ID for the storage account. Now let's go ahead and save and close all of these files. We'll run terraform format and terraform validate to ensure that our files are set up correctly. All right, we have a valid configuration. Now, let's push this into source control. And you're already familiar with this process from the previous lessons, so I'll skip over this and we'll come back when it's done. All right, that code's been pushed to the remote GitHub repository. Let's go to GitHub and we'll create a tagged release, and then we can jump over to Terraform and publish the module to the private registry. First we'll select tags. We'll select to create a new release. We'll create a tag and we'll name it 1.0 0. And I'm just going to use the version for the release title. And then we go ahead and publish the release. Now, let's jump over to Terraform Cloud and publish this module to the private registry. I'll click Registry. Then I'll select the option to publish a module. I've already connected to GitHub for version control, so I'll go ahead and select GitHub. And it's not listing the repository here, so I'm going to go ahead and type the ID for the repository manually. And then we can go ahead and publish our module. All right, our module is now available in our private registry. Now let's go ahead and author and apply a configuration using the module. We'll go back to Visual Studio Code. Now to save some time, I'm going to skip over the process of creating a new working directory, main.tf file, and adding the Terraform configuration and provide a configuration as we're already well and truly familiar with that process by now. Now we need credentials to access the private registry. If we were to integrate this configuration with Terraform Cloud using the cloud block in our Terraform configuration block, we could skip this step, but I'm going to go ahead and just save credentials locally in a terraform.rc file. Firstly, I'll grab the credentials block from Terraform Cloud, and I'll create a new file. I'll just create a text file. I'll paste in that block, and I'll get a valid API token from Terraform Cloud. And this is something we've also done in the previous lessons, so I'll go and grab that, paste it in here, and we'll come back when I'm done. Now, we'll go ahead and save this file. It needs to be placed in APPDATA, and it needs to be named terraform.rc. We'll go ahead and save that file now. We can go ahead and close that credential file now. Now, I'm going to deploy to an existing resource group, so I'll declare and import that resource group, and I'll come back when that's done. We can now go ahead and define the module block. And I'm not going to write that manually because Terraform Cloud gives me the code for that. So let's jump over to Terraform Cloud. We'll navigate into our registry, select that module, copy the code, and we'll paste that in Visual Studio Code. Now, we'll save that file, and we'll run terraform init to download that required module. And then we'll provide the required arguments for our module, including the resource group name, the location, and you can see, I'm just using tab to complete these so I don't have to type these out in full. I'll set the environment to be production. And then we'll provide our storage account name. And I'll just provide a unique name. Go ahead and save that file. We'll run terraform validate. And now we'll run terraform apply, which will automatically run a plan for us. We'll select yes to deploy. All right, let's take a look at our storage account in the Azure portal. And we can see our storage account, and we can see it's using geo-redundant storage, and it has an environment tag set to production, just as we configured in our module. All right, that concludes this demo. All right, let's wrap up with a quick summary and some key takeaways. First up, we learned that you can use Terraform variables to make your code more flexible and reusable and reduce the need to repeat code in your configuration. Variables include input variables, locals, and output variables. We also took a look at how you can create Terraform modules to create reusable resource configurations. And we learned that you can use community created modules from the public Terraform registry, or you can upload your modules to a private registry in Terraform Cloud to share the modules you've created with your team. There's a hands-on lab immediately after this lesson that you can do to apply the concepts we've covered in this lesson. But that's all from me for now. Thanks for being with me. I hope you found this useful, and I look forward to seeing you in the next lesson!

Course Conclusion

Hello, Cloud Gurus, and welcome back! In this lesson, we're going to wrap up all of the topics we've covered in this course with a bit of a summary of what we've covered and how that has prepared you for tackling the final challenge lab in this course. Now, we've covered a lot of ground in this short course. Let's recap what we have covered and how it has led us to where we are now. We started the course with a general introduction to the Terraform workflow and using the Terraform command-line interface. And we explored the challenges we're planning to tackle in this course. And those challenges revolved around River City AI and their requirements. And those requirements have really centered around two pillars, the first being security and the other being agility, which are two really common requirements for many organizations when they are deploying cloud workloads. River City AI's first requirement was to make sure that it had secure secrets. And we learned that Terraform stores sensitive values in a local state file by default. To solve this problem, we went through the process of securing the state by moving to remote state in Terraform Cloud, which also solved another problem we needed to address in relation to agility. And that is that other platform engineers will need to access the state file so infrastructure can be deployed and updated faster by multiple people. But that only solved half of the collaboration problem. We still had our Infrastructure as Code stored locally on our workstation. To solve the code collaboration problem, we implemented a GitHub repository as our version control system and moved our configuration into the repository, and this allowed us to use things like branches to protect our production infrastructure from unintended code changes. And moving our code to GitHub also solved a couple of other problems we had, including continuous delivery and the ability to use an extensible delivery pipeline. GitHub Actions provided this functionality for us and integrated really well with Terraform Cloud and our GitHub repository. Now, River City AI also had the requirement for reusable code, which we learned can be solved by using Terraform variables to increase the reusability of our code and modules to allow us to share the infrastructure we've defined as code. But River City AI also wanted the modules to be secure and private. To meet both of these requirements, we learned how to create reusable Terraform modules and store them in a private registry in Terraform Cloud. And this used tagged releases in GitHub to control the publication of new module versions to that private registry. Now, hopefully you followed along and completed each of the hands-on labs as we went through the lessons. Well, now it's your turn. I have one final challenge for you. After completing this lesson, there is a final challenge lab that will see how you go at completing each of the steps we've completed in this course so far. And with that, thanks for being with me. I hope you've enjoyed the course and learned something new. I've sure had a great time creating this course, as there is just so much good stuff in such a short course. Now, good luck with the final challenge lab, and I look forward to seeing you in another cloud adventure again real soon!