**Plan provisioning tasks for Azure Pipelines**

At this point, you have a basic Terraform plan that you can run from the command line.

However, when you add a provisioning step to Azure Pipelines, Terraform needs permission to make infrastructure changes on your behalf. You don't want to check your credentials in to source control, so you need a better plan.

Also, Terraform maintains a state file to help it understand how your infrastructure configuration maps to active Azure resources. When your pipeline exits, the agent and its environment are destroyed and you will lose your Terraform state file.

In this part, you learn how Azure Blob storage and service principals enable you to run your Terraform plan from Azure Pipelines.

**What else do we need?**

Andy and Tim are taking a short break to discuss how they can run their Terraform plan from Azure Pipelines. Let's listen in.

**Tim:** I'm starting to see how this all works. Are we ready to run our Terraform plan from Azure Pipelines?

**Andy:** Almost. I see two questions that we need to answer first:

* How do we maintain the Terraform state file?

Our existing Terraform plan maintains its state file locally in Azure Cloud Shell. How can we persist the state file so that multiple pipeline runs can share it? If we don't persist the state file, the agent that performs the next pipeline run won't have one that it can access.

Also, how can we ensure that the state file properly *locked*? In other words, how do we ensure that only one pipeline run can access the state file at a time?

* How do we authenticate access to Azure from Azure Pipelines?

Our existing pipeline uses a service connection to deploy the *Space Game* website to Azure App Service. The deployment task, AzureWebApp@1, understands how to work with the service connection. But a service connection does not enable us to authenticate arbitrary commands with Azure, such as terraform apply.

Tim and Andy consider a few options. Each approach comes with a few drawbacks. After doing some more research, they come up with a plan.

**Andy:** I discovered that we can use what's called a *back end* to store the state file remotely. Terraform supports many kinds of back ends, such as an Artifactory repository or a Consul key value (Console KV) store. But there's also built-in support for Azure Blob storage. Here, all we need is an Azure storage account and a Blob storage container. We specify the storage account and the storage container in our Terraform plan, and Terraform manages the state file for us.

**Tim:** That's great news! I also discovered how to authenticate access to Azure from Azure Pipelines. When running Terraform through automation, it's common to set up a *service principal* that can authenticate access to Azure. Setting up our own service principal is similar to how a service connection works in Azure Pipelines. We create a service principal and then export a few environment variables when the pipeline runs.

**Andy:** Terrific! Let's make sure we understand everything we need before we move forward.

**What's Azure Storage?**

Azure provides many ways to store your data. Azure selected four foundational data services and placed them together under the name *Azure Storage*. The four services are Azure Blob storage, Azure Files, Azure Queue storage, and Azure Table storage.

**What's a storage account?**

A *storage account* is a container that groups a set of Azure Storage services together. A storage account can include resources from Azure Blob storage, Azure Files, Azure Queue storage, and Azure Table storage.

**What's Azure Blob storage?**

Azure Blob storage is an object storage solution that can store large amounts of unstructured data. A file, such as the Terraform state file, is an example of unstructured data.

Terraform can manage the state file for you through Azure Blob storage. It also locks access to this file so that only one Terraform run can access the state file at a time. Doing so ensures that simultaneous Terraform runs don't interfere with one another.

**What's a service principal?**

A *service principal* is an identity with a limited role that can access Azure resources. Think of a service principal as a service account that can perform automated tasks on your behalf.

**What information does Terraform need?**

There are a few ways to run your Terraform plan as a service principal. For example, you can provide the details about your service principal directly in your plan file.

Another way is to export environment variables that Terraform understands. These environment variables are:

* ARM\_CLIENT\_ID

This is the name that your service principal uses to sign in to Azure.

* ARM\_CLIENT\_SECRET

This is the password that your service principal uses to sign in to Azure.

* ARM\_TENANT\_ID

A tenant is the organization in Azure Active Directory (Azure AD) where your service principal is located.

* ARM\_SUBSCRIPTION\_ID

This is your Azure subscription ID.

Each of these variables specifies secret information that you don't want to share with others. To help ensure that the pipeline can access these variables in a secure way, you'll use secret variables to store these values in Azure Pipelines.

**What other considerations do I need to make?**

The Tailspin team needs to make a few considerations before it can run Terraform from Azure Pipelines:

* Ensuring that secret values, such as service principal credentials, are not logged or written to the output stream
* Ensuring that Terraform does not prompt for missing input values
* Passing output from one job to the next job

**How do I protect secrets in my pipeline tasks?**

To have Terraform provision resources under your service principal account, recall that you need to export these environment variables:

* ARM\_CLIENT\_ID
* ARM\_CLIENT\_SECRET
* ARM\_TENANT\_ID
* ARM\_SUBSCRIPTION\_ID

In this module, you'll add each of these to your project as pipeline variables. Your pipeline tasks can access variables, but these variables are not exported as environment variables for use by terraform or other child processes.

Although you can export these variables from your Bash script, keep in mind that these variables are encrypted as secret variables in your pipeline. Operating systems often log commands for the processes that they run. You wouldn't want the log to include a secret that you passed in as an input.

The recommended way to pass secrets like this to a Bash or PowerShell script is to specify them in your script's environment. Here's an example:

ymlCopy

env:

ARM\_CLIENT\_ID: $(ARM\_CLIENT\_ID)

ARM\_CLIENT\_SECRET: $(ARM\_CLIENT\_SECRET)

ARM\_TENANT\_ID: $(ARM\_TENANT\_ID)

ARM\_SUBSCRIPTION\_ID: $(ARM\_SUBSCRIPTION\_ID)

This env section maps each pipeline variable to a Bash or PowerShell variable with the same name. You'll see a more complete example later in this module.

**How do I ensure that Terraform does not prompt for input values?**

Recall that a Terraform variable can have a default value. Here's an example that sets the default resource group location to "westus":

TerraformCopy

variable "resource\_group\_location" {

default = "westus"

description = "The location of the resource group"

}

However, you don't have to provide a default value. For example:

TerraformCopy

variable "resource\_group\_location" {

description = "The location of the resource group"

}

When you don't provide a default value, and you don't provide the -var argument on the command line or provide a value in a *.tfvars* file, Terraform prompts you for the value.

What happens when this occurs in the pipeline? You're not able to enter a value because the pipeline does not provide interactivity. So, the pipeline will stall.

The answer is to provide the -input=false argument to the terraform init and terraform apply commands. This argument tells Terraform to exit with an error status if there are any variables whose values could not be set. This prevents the pipeline from stalling forever while waiting for user input.

That way, if you make changes to your Terraform plan but forget to set a variable's value, the pipeline simply fails. From here, you can specify the variable's value and push the change through the pipeline a second time.

**How can I pass output from one job to the next job?**

For learning purposes, your Terraform plan uses a random number to make your App Service name unique. In practice, you would choose a name that matches the name of your application or service.

Recall from the Deploy applications with Azure DevOps  learning path that the AzureWebApp@1 task needs the name of the App Service instance where you want to deploy your application.

ymlCopy

- task: AzureWebApp@1

displayName: 'Azure App Service Deploy: website'

inputs:

azureSubscription: 'Resource Manager - Tailspin - Space Game'

appName: '$(WebAppNameDev)'

package: '$(Pipeline.Workspace)/drop/$(buildConfiguration)/\*.zip'

In the previous learning path, you set the WebAppNameDev pipeline variable because you knew the name of your App Service instance in advance. Now that you're provisioning from the pipeline, you don't know this name until after the provisioning step finishes. That's because Terraform generates the random number the first time it provisions your infrastructure.

After the provisioning step happens in one job, how can you pass the name to the deployment step, in another job?

One answer is to use the ##vso[] syntax to write the name to the pipeline as a pipeline variable.

Later in this module, you'll use a Bash script to run Terraform from a pipeline task. After Terraform runs, you run terraform output to get the name of the App Service instance. Here's an example:

BashCopy

# Get the App Service name for the Dev environment.

WebAppNameDev=$(terraform output appservice\_name\_dev)

That command assigns the result to a Bash variable named WebAppNameDev. Next, you write the value to the pipeline by using the ##vso[] syntax. Here's an example:

BashCopy

# Write the WebAppNameDev variable to the pipeline.

echo "##vso[task.setvariable variable=WebAppNameDev;isOutput=true]$WebAppNameDev"

That command writes a variable to the pipeline with the same name, WebAppNameDev. This syntax is a way to dynamically write variables back to the pipeline so that another process can read them. The isOutput=true part enables another job to read the variable. When you omit the isOutput=true attribute, only tasks within the same job can access the variable.

To read the variable from the pipeline, the deployment job defines a variables section that reads the WebAppNameDev variable from the pipeline. Here's an example:

ymlCopy

- deployment: Deploy

dependsOn: Provision

variables:

WebAppNameDev: $[ dependencies.Provision.outputs['RunTerraform.WebAppNameDev'] ]

pool:

vmImage: 'ubuntu-18.04'

The part in brackets loads the WebAppNameDev variable from the task named RunTerraform in the Provision job. You'll see a complete example shortly.

**The plan**

**Tim:** Here's what I propose we do next, before we run Terraform from the pipeline:

1. Set up Blob storage in Azure.
2. Connect the plan to Blob storage so that the state file is maintained remotely.
3. Set up the service principal.
4. Export environment variables that enable Terraform to run the plan as the service principal.

Let's also modify the Terraform plan to define an App Service instance that maps to our **dev** environment. Later, we can add additional App Service instances for our **test** and **staging** environments.

**Andy:** Let's get to work!