**How does Terraform work?**

* 10 minutes

Andy and Tim have decided to use Terraform to provision their infrastructure. To prepare for the exercises in this module, let's examine how Terraform works and see some of its most commonly used features.

In this section, you learn:

* How Terraform files are structured.
* What parts make up a typical Terraform plan.
* How Terraform commands map to your infrastructure's lifecycle.

**How are Terraform files structured?**

When you're working with Terraform code, you commonly use these files:

* *main.tf*

A *.tf* file holds your Terraform configuration code. This file is often called a Terraform *plan*. Your Terraform plan specifies the infrastructure resources that you need.

You can name this file whatever you want, but it's often named *main.tf*.

* *terraform.tfvars*

Your Terraform plan can include variables that are set when the plan runs. Variables enable you to express your configuration more generally.

For example, you might specify the region, or location, of your resources as a variable. When you run your plan, you specify the region, such as "westus" or "northeurope".

Variables enable you to more easily reuse your Terraform code without the need to modify the source code. They also enable you to more easily change how multiple resources are configured, because variables are defined in one location.

There are a few ways to specify each variable's value when the plan runs. You'll learn about those shortly. A *.tfvars* file is a good way to maintain larger sets of variables.

You can name this file whatever you want, but this file is often named *terraform.tfvars*.

* *terraform.tfstate*

*terraform.tfstate* is called the *state file*. It's a JSON file that Terraform manages. It helps map the resources that you define in your plan to the running resources that your plan produces.

For example, you might include a name as part of your resource description. But the running resource is identified by a GUID, URI, or numeric ID. Terraform uses the state file to map your name to how Azure or another platform identifies the same resource.

In this module, you start by running Terraform locally from Azure Cloud Shell. Here, Terraform manages the state file locally in your Cloud Shell session. Later, you'll move the state file to an Azure storage account to provide a shared, central location for Terraform to use in Azure Pipelines.

You'll work with each of these files shortly.

**What's in a Terraform plan?**

Here, we define some common pieces of your Terraform plan.

**Providers**

A *provider* links the Terraform resources that you specify to the API calls that are needed to provision and manage those resources.

In terms of declarative programming, think of your plan as *what* infrastructure you need. Think of a provider as *how* your plan is mapped to running cloud resources.

Recall that Terraform supports a number of public clouds and private cloud frameworks. Think of the terraform program as the core Terraform engine. The functionality for each provider is defined in a plug-in. A provider might be Azure, Docker, or Amazon Web Services, for example.

Provider software versions are maintained separately from the terraform program. You can specify the provider version in a provider block. Here's an example:

TerraformCopy

provider "azurerm" {

version = ">=2.0.0"

features {}

}

When working with Azure resources, the provider block is required in your plan. When you're getting started, you can use the most basic settings as shown here. In practice, you might specify additional features to customize the behavior of certain resources. For example, you can specify whether the OS disk should also be destroyed when a virtual machine is destroyed.

**Variables**

Earlier, we defined a *variable* in Terraform as a way to express your configuration more generally. Variables are set when the plan runs.

Here's an example of a variable block that defines the location, or region, for an Azure resource group:

TerraformCopy

variable "resource\_group\_location" {

description = "The location of the resource group"

}

Optionally, you can provide a default value. This example uses "westus":

TerraformCopy

variable "resource\_group\_location" {

default = "westus"

description = "The location of the resource group"

}

When you define your resource group, you use the var.variable syntax to read your variable. Here's an example:

TerraformCopy

resource "azurerm\_resource\_group" "my" {

name = var.resource\_group\_name

location = var.resource\_group\_location

}

To specify the values of your variables when Terraform runs, you can use:

* The -var argument.
* A *.tfvars* file.
* Environment variables.
* A Terraform Cloud workspace.

If you don't provide a variable's value, and the variable doesn't have a default value, Terraform prompts you for the value when your plan runs. This approach works well when you're getting started, but it doesn't work well in CI/CD pipelines.

**Resources**

A *resource* is the primary construct in Terraform. A resource defines one piece of infrastructure that you need. Examples of compute resources include virtual machines and Azure App Service. Examples of data and storage resources include Azure Storage and Azure SQL Database. There are many more types of resources.

Here are a few resources you'll work with shortly. These resources define an Azure resource group, an App Service plan, and an App Service instance.

TerraformCopy

variable "app\_service\_name\_prefix" {

default = "my-appsvc"

description = "The beginning part of the app service name"

}

resource "random\_integer" "app\_service\_name\_suffix" {

min = 1000

max = 9999

}

resource "azurerm\_resource\_group" "my" {

name = var.resource\_group\_name

location = var.resource\_group\_location

}

resource "azurerm\_app\_service\_plan" "my" {

name = var.app\_service\_plan\_name

location = azurerm\_resource\_group.my.location

resource\_group\_name = azurerm\_resource\_group.my.name

kind = "Linux"

reserved = true

sku {

tier = "Basic"

size = "B1"

}

}

resource "azurerm\_app\_service" "my" {

name = "${var.app\_service\_name\_prefix}-${random\_integer.app\_service\_name\_suffix.result}"

location = azurerm\_resource\_group.my.location

resource\_group\_name = azurerm\_resource\_group.my.name

app\_service\_plan\_id = azurerm\_app\_service\_plan.my.id

}

Notice that you can define some attributes, such as the parent resource group's name and location, directly from other resources. Terraform creates a dependency graph that expresses these relationships so that it can create and manage your resources in the correct order.

**Output values**

An *output value* is a way to expose information to the person or process that runs your plan.

Terraform prints each output value after it applies your plan. You can also fetch an output value later when you need it.

Here's an example that outputs the host name of an App Service instance:

TerraformCopy

output "website\_hostname" {

value = azurerm\_app\_service.my.default\_site\_hostname

description = "The hostname of the website"

}

You can fetch output values after Terraform runs. Later, you'll see how to fetch the name of an App Service instance whose name is generated dynamically. You'll pass that name to a later job in Azure Pipelines.

**Terraform settings**

A *Terraform settings* block specifies configuration settings for Terraform itself.

You commonly use this block to specify the version of Terraform that you need. Here's an example:

TerraformCopy

terraform {

required\_version = "> 0.12.0"

}

Specifying the version enables you to *pin*, or specify, the version you've tested with. As Terraform releases new versions, you can test and verify your configurations and then update the version when you're ready.

The settings block also enables you to specify the location of the state file. You'll do so later when you move the state file to an Azure storage account.

**What commands will I need?**

Terraform provides commands that map to your infrastructure's lifecycle. Here are some common commands that you'll use in this module.

**Initialize**

The terraform init command initializes your Terraform environment. This command downloads the plug-ins that you need. It also verifies that Terraform can access your plan's state file.

**Plan**

The terraform plan command produces an execution plan that's based on your configuration. This command doesn't modify any infrastructure. It's just a way for a human to review what changes will be made if the plan is applied.

This command is useful for developing your Terraform configuration because it helps you visualize and understand the effect that your plan will have.

The output resembles how many file difference tools work. You see a plus (**+**) symbol next to resources that Terraform will create. You see a minus (**-**) symbol next to resources that Terraform will destroy. The output also shows each attribute that's applied to each resource, such as the resource group's name or the storage account tier.

You typically omit this command when running Terraform in a CI/CD pipeline. By the time your plan reaches the pipeline, your plan should express your infrastructure requirements and you should understand the effect that your plan will have.

**Apply**

The terraform apply command runs your execution plan. Think of it as a way to apply the proposed changes that you get from the terraform plan command.

Like terraform plan, this command prints out an execution plan. It also prompts you to verify the configuration one final time before the plan runs. Later, you'll see how to skip this verification step when you run your plan in Azure Pipelines.

The terraform apply command is an idempotent operation. Recall that an idempotent operation is one that that provides the same result each time you apply it. This means that terraform apply takes no action if the state of your infrastructure matches what's expressed in your plan. Terraform won't re-create or duplicate your resources if the environment has not changed. This makes terraform apply safe to run in a CI/CD pipeline.

If your Terraform plan changes or some other process inadvertently changes your infrastructure, terraform apply places your infrastructure in the desired state.

**Destroy**

The terraform destroy command destroys all infrastructure resources that are defined in your plan.

Like terraform apply, the terraform destroy command prompts you to verify the operation and provides an execution plan that you can review before you take action.

You might not run terraform destroy in your pipeline if your resources are long lived. But you'll likely run this command many times as you develop your plan and test out new ideas.