**Exercise - Run a basic Terraform plan from Azure Cloud Shell**

In this exercise, you run a Terraform plan that provisions Azure App Service.

During the process, you:

* Walk through a basic Terraform plan so that you understand what each piece does.
* Create the variables file to hold the Azure region that you choose.
* Provision your infrastructure and verify the result.
* Run the plan a second time to see idempotency in action.
* Tear everything down.

**Important**

To complete this module, you need your own **Azure subscription**. Get started for free.

**Open Cloud Shell through the Azure portal**

Open Azure Cloud Shell through the Azure portal so that you can work with Terraform code and run your configuration. We use Cloud Shell here because it comes with Terraform already set up for you.

You can also install and run Terraform locally from a terminal or a PowerShell window. Later, you'll run Terraform from Azure Pipelines.

1. Go to the Azure portal  and sign in.
2. From the menu bar, select **Cloud Shell**. When you're prompted, select the **Bash** experience.

Selecting Cloud Shell from the menu bar

**Note**

Cloud Shell requires an Azure Storage resource to persist any files that you create while working in Cloud Shell. When you first open Cloud Shell, it offers to create a resource group, storage account, and Azure Files share on your behalf. This is a one-time step and will be automatically attached for all future Cloud Shell sessions.

**Create a working directory**

Create a directory to hold your Terraform plan. Doing so helps keep your Terraform code separate from your other work.

1. In Cloud Shell, create a directory named *mslearn-terraform-local-state*.

**Bash**

mkdir ~/mslearn-terraform-local-state

1. Move to the *mslearn-terraform-local-state* directory.

**Bash**

cd ~/mslearn-terraform-local-state

**Add the Terraform plan**

Create a basic Terraform plan in a file named *main.tf*.

1. From the *mslearn-terraform-local-state* directory, open the editor.

**Bash**

code main.tf

1. Add the following code to *main.tf*, and then save the file.

**Terraform**

terraform {

required\_version = "> 0.12.0"

}

provider "azurerm" {

version = ">=2.0.0"

features {}

}

variable "resource\_group\_name" {

default = "my-rg"

description = "The name of the resource group"

}

variable "resource\_group\_location" {

default = "westus"

description = "The location of the resource group"

}

variable "app\_service\_plan\_name" {

default = "my-asp"

description = "The name of the app service plan"

}

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

}

output "website\_hostname" {

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

description = "The hostname of the website"

}

**Understand the Terraform plan**

Here, you briefly walk through each section of the Terraform plan to better understand how it works.

**Terraform settings**

The plan starts with a terraform block that specifies the version of Terraform that your plan needs.

**Terraform**

terraform {

required\_version = "> 0.12.0"

}

The required\_version attribute specifies version 0.12.0 or later. In practice, you might use the = syntax to pin a specific version.

**Azure provider settings**

The plan doesn't need to control specific Azure resource features, so it uses a basic provider block. The version attribute specifies version 2.0.0 or later of the Azure provider. In practice, you might use the = syntax to pin a specific version.

**Terraform**

provider "azurerm" {

version = ">=2.0.0"

features {}

}

**Terraform variables**

The plan includes these four variables:

* The resource group's name
* The resource group's location or region
* The name of the App Service plan
* A prefix that's later added to the name of the App Service instance

**Terraform**

variable "resource\_group\_name" {

default = "my-rg"

description = "The name of the resource group"

}

variable "resource\_group\_location" {

default = "westus"

description = "The location of the resource group"

}

variable "app\_service\_plan\_name" {

default = "my-asp"

description = "The name of the app service plan"

}

variable "app\_service\_name\_prefix" {

default = "my-appsvc"

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

}

Each variable provides a default value. Shortly, you'll create a *.tfvars* file that specifies the value for the resource group's location.

**The random\_integer resource**

We haven't yet introduced the random\_integer resource. It's a resource that generates a random whole number that's between the specified minimum and maximum values.

**Terraform**

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

min = 1000

max = 9999

}

The plan uses this value to add a unique number to the name of your App Service instance. Azure uses the name of your App Service instance to form its host name. The name must be unique. In practice, you would specify a name for your App Service instance that describes your application.

When Terraform first runs this plan, it writes the generated value to the state file. So you get the same number each time Terraform runs.

**The resource group**

The azurerm\_resource\_group resource creates the resource group that holds your App Service plan and App Service instance.

**Terraform**

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

name = var.resource\_group\_name

location = var.resource\_group\_location

}

To specify the resource group's name and location, the plan uses the var.variable syntax to read the variables defined earlier in the plan.

The name "my" enables you to refer to this resource in other parts of your plan. This name does not appear in your Azure resource. Think of it as you would a variable in any other programming language.

**The App Service plan**

The azurerm\_app\_service\_plan resource defines the App Service plan. The App Service plan is set to run on Linux and uses the **Basic** pricing tier, which is intended for apps that have lower traffic requirements.

**Terraform**

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"

}

}

As with the resource group, this resource uses the var.variable syntax to specify the name of the App Service plan.

To specify the App Service plan's location and parent resource group name, this resource reads attributes from the azurerm\_resource\_group resource named "my." Doing so enables this resource to remain synchronized with its parent resource group if the parent resource group changes.

**The App Service instance**

The azurerm\_app\_service resource defines the App Service instance.

**Terraform**

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

}

Like the azurerm\_app\_service\_plan resource, this resource reads the location and parent resource group name from the azurerm\_resource\_group resource named "my." It reads the App Service plan ID from the azurerm\_app\_service\_plan resource.

Azure uses the App Service name to form the host name. To specify the App Service name, the plan combines (or *interpolates*) the prefix variable and the variable that holds the name of the App Service instance.

**The host name output value**

In this example, you don't know the name or host name of the App Service instance until the plan runs. That's because the plan uses a random integer as part of the name.

The output block prints the host name of the App Service instance after the plan runs. Doing so gives you a URL that you can test to verify that App Service is running.

**Terraform**

output "website\_hostname" {

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

description = "The hostname of the website"

}

**Tip**

In Cloud Shell, you can close the editor now if you want. But leave the command window open for the next part.

**Select an Azure region**

A *region* is one or more Azure datacenters within a specific geographic location. East US, West US, and North Europe are examples of regions. Every Azure resource, including an App Service instance, is assigned a region.

To make the commands easier to run, start by selecting a default region. After you specify the default region, later commands use that region unless you specify a different region.

1. From Cloud Shell, run the following az account list-locations command to list the regions that are available from your Azure subscription.

**Azure CLI**

az account list-locations \

--query "[].{Name: name, DisplayName: displayName}" \

--output table

1. From the **Name** column in the output, choose a region that's close to you. For example, use **eastasia** or **westus2**.
2. Run the following read command to set a Bash variable that's named AZ\_LOCATION.

**Bash**

read AZ\_LOCATION

At the prompt, enter the region that you chose in the previous step.

This Bash variable makes it easier to run the command that comes next. In practice, this Bash variable isn't required.

1. Print the Bash variable to verify that it was set correctly.

**Bash**

echo $AZ\_LOCATION

You see your region in the results.

**Create the variables file**

Create the Terraform variables file, *terraform.tfvars*. In the file, you add the Azure region that you just chose.

This file can also contain values for your other variables. But for learning purposes, here you only need to specify the region.

1. Run the following command. It creates a file named *terraform.tfvars* and sets the contents with the value for the resource\_group\_location Terraform variable.

**Bash**

echo 'resource\_group\_location = "'$AZ\_LOCATION'"' | tee terraform.tfvars

1. Print *terraform.tfvars* to confirm the location.

**Bash**

cat terraform.tfvars

The output resembles this:

**Output**

resource\_group\_location = "northeurope"

**Provision your infrastructure**

Initialize Terraform, examine the proposed execution plan, and then apply the plan.

1. Run terraform init to initialize your Terraform environment.

**Bash**

terraform init

This command downloads the latest versions of the "azurerm" and "random" provider plug-ins. If you need a specific version of either of these plug-ins, you include a provider block in your Terraform plan.

1. Run terraform plan to view Terraform's proposed execution plan.

**Bash**

terraform plan

The command reads the resource\_group\_location variable from *terraform.tfvars*.

1. Examine the plan.

You see that the plan includes the resource group, the random number, the App Service plan, and the App Service instance. Many attributes, including the value of the random number, can't be known until Terraform runs the plan.

Because you haven't yet run this plan, each resource is marked with a plus (**+**) symbol to indicate that it would be created.

In practice, you would verify that this plan meets your infrastructure requirements.

1. Run terraform apply to run the plan.

**Bash**

terraform apply

When you're prompted, enter **yes**.

Later, you'll see how to automatically apply the configuration when you run the plan in Azure Pipelines.

**Important**

If you don't finish this exercise, be sure to run the terraform destroy command shown here to ensure that you're not charged for Azure resources you no longer need.

**Verify the result**

After the plan finishes, Terraform writes the output value that contains the host name for your site. But let's see how to get this value a second time when you need it later.

1. Run terraform output to print the host name.

**Bash**

terraform output

The output resembles this:

**Output**

website\_hostname = my-appsvc-8499.azurewebsites.net

1. Run the following command to print only the host name.

**Bash**

terraform output website\_hostname

The output resembles this:

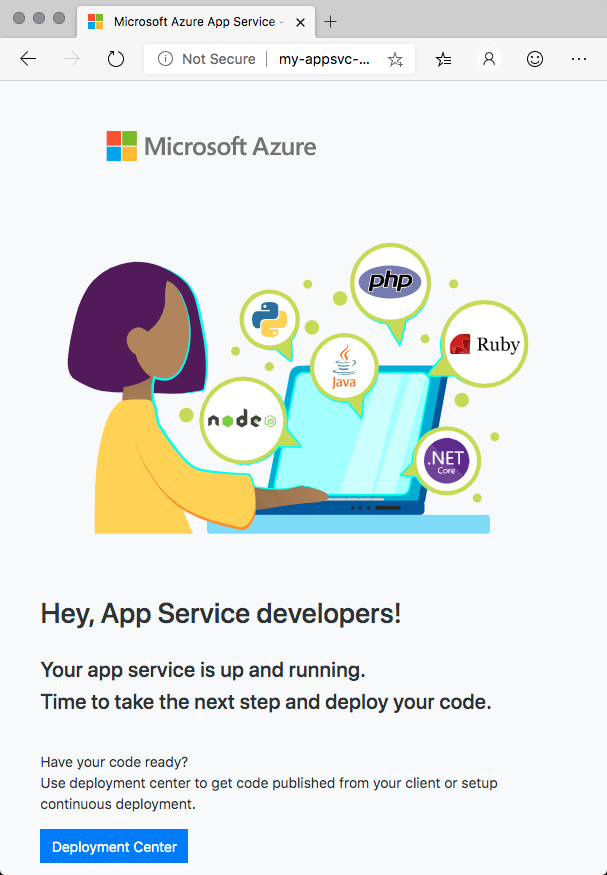
**Output**

my-appsvc-8499.azurewebsites.net

This form is useful when you have many output values, but you're interested in only one of them. It's also a useful way to pass values to other commands because this form prints only the value. You'll see an example that uses this form later.

1. On a new browser tab, go to your site.

You see the default App Service home page:



Later, you'll deploy the *Space Game* website to App Service from Azure Pipelines.

1. Run the following az webapp list command. This command verifies that App Service is running and that the correct parent resource group contains it.

**Azure CLI**

az webapp list --resource-group my-rg --output table

The output resembles this:

**Output**

Name Location State ResourceGroup DefaultHostName AppServicePlan

-------------- ------------ ------- --------------- -------------------------------- ----------------

my-appsvc-8499 North Europe Running my-rg my-appsvc-8499.azurewebsites.net my-asp

**Examine the state file**

You don't typically work with the Terraform state file directly, but here you examine it to get a sense of what it contains.

Run the following command to print the contents of your Terraform state file, *terraform.tfstate*:

**Bash**

cat terraform.tfstate

Take a moment to examine the output. Although Terraform always checks with Azure to get the current state of your resources before it applies changes, the state file can help Terraform map your plan to your running resources. In this example, the state file holds the value of your random number. This value is not stored in Azure.

**Run the plan a second time**

Terraform is idempotent. That means you can run it as many times as you want. Terraform applies infrastructure changes only when your plan changes or the state of your Azure resources doesn't match what's in your plan.

1. Run terraform plan a second time.

**Bash**

terraform plan

This time, you see that Terraform proposes no changes. That's because your Terraform plan did not change and you haven't modified any of your Azure resources.

1. Run terraform apply a second time.

**Bash**

terraform apply

As you would expect, Terraform does not apply any changes. Later, you'll see the benefit of this behavior when you provision your infrastructure in Azure Pipelines.

**Destroy your infrastructure**

You're finished with the infrastructure that you provisioned. Here, you destroy the infrastructure that Terraform is managing for you.

1. Run terraform destroy to destroy your resources.

**Bash**

terraform destroy

When you're prompted, enter **yes**.

Much like terraform plan, the terraform destroy command gives you a proposed plan of action. Terraform marks each resource with a minus (**-**) symbol to indicate that it would be destroyed.

1. Run the following az group list command to verify that your Azure subscription no longer contains a resource group named "my-rg."

**Azure CLI**

az group list \

--query "[?contains(@.name, 'my-rg')].{name: name}" \

--output tsv

This command produces no output, which verifies that the resource group no longer exists.

**What's next?**

Let's see what Tim and Andy plan to do next.

**Tim:** That was easy. So we can just put this in our pipeline and have it create the resources?

**Andy:** This is a great start, but we have a few things to consider. Let's do that next.