

# Contents

[Terraform on Azure documentation](#)

[Overview](#)

[About Terraform on Azure](#)

[Quickstarts](#)

[Install and configure Terraform](#)

[Create a Terraform configuration](#)

[Create a Linux VM](#)

[Tutorials](#)

[Setup and configuration](#)

[Store Terraform state in Azure Storage](#)

[Configure Azure Cloud Shell for Terraform](#)

[App Service](#)

[Terraform and Azure provider deployment slots](#)

[Containers](#)

[Create a Kubernetes cluster with AKS](#)

[Create an Application Gateway ingress controller in AKS](#)

[Infrastructure](#)

[Create a VM cluster with Terraform modules](#)

[Create a VM cluster with Terraform and HCL](#)

[Provision VM scale set with infrastructure](#)

[Provision VM scale set from a Packer custom image](#)

[Networks](#)

[Hub and spoke topology](#)

- [1. Create hub and spoke topology](#)
- [2. Create on-premises virtual network](#)
- [3. Create hub virtual network](#)
- [4. Create hub virtual network appliance](#)
- [5. Create spoke network](#)
- [6. Validate network topology connectivity](#)

## Tools

[Install the Terraform Visual Studio Code extension](#)

[Create a Terraform base template using Yeoman](#)

[Test Terraform modules using Terratest](#)

## Samples

[Configuration templates](#)

## Reference

[Azure module registry](#)

[Terraform Azure Provider](#)

## Resources

[Azure & Terraform](#)

[Azure Roadmap](#)

[Hashicorp Terraform](#) is an open-source tool for provisioning and managing cloud infrastructure. It codifies infrastructure in configuration files that describe the topology of cloud resources. These resources include virtual machines, storage accounts, and networking interfaces. The Terraform CLI provides a simple mechanism to deploy and version the configuration files to Azure.

This article describes the benefits of using Terraform to manage Azure infrastructure.

## Automate infrastructure management.

Terraform's template-based configuration files enable you to define, provision, and configure Azure resources in a repeatable and predictable manner. Automating infrastructure has several benefits:

- Lowers the potential for human errors while deploying and managing infrastructure.
- Deploys the same template multiple times to create identical development, test, and production environments.
- Reduces the cost of development and test environments by creating them on-demand.

## Understand infrastructure changes before being applied

As a resource topology becomes complex, understanding the meaning and impact of infrastructure changes can be difficult.

The Terraform CLI enables users to validate and preview infrastructure changes before application. Previewing infrastructure changes in a safe manner has several benefits:

- Team members can collaborate more effectively by quickly understanding proposed changes and their impact.
- Unintended changes can be caught early in the development process

## Deploy infrastructure to multiple clouds

Terraform is adept at deploying an infrastructure across multiple cloud providers. It enables developers to use consistent tooling to manage each infrastructure definition.

## Next steps

Now that you have an overview of Terraform and its benefits, here are suggested next steps:

- Get started by [installing Terraform and configuring it to use Azure](#).
- [Create an Azure virtual machine using Terraform](#)
- Explore the [Azure Resource Manager module for Terraform](#)

Terraform provides an easy way to define, preview, and deploy cloud infrastructure by using a [simple templating language](#). This article describes the necessary steps to use Terraform to provision resources in Azure.

To learn more about how to use Terraform with Azure, visit the [Terraform Hub](#).

#### NOTE




For Terraform specific support, please reach out to Terraform directly using one of their community channels:

- The [Terraform section](#) of the community portal contains questions, use cases, and useful patterns.
- For provider-related questions please visit the [Terraform Providers](#) section of the community portal.

## Use Azure Cloud Shell

Azure hosts Azure Cloud Shell, an interactive shell environment that you can use through your browser. You can use either Bash or PowerShell with Cloud Shell to work with Azure services. You can use the Cloud Shell preinstalled commands to run the code in this article without having to install anything on your local environment.

To start Azure Cloud Shell:

OPTION	EXAMPLE/LINK
Select <b>Try It</b> in the upper-right corner of a code block. Selecting <b>Try It</b> doesn't automatically copy the code to Cloud Shell.	
Go to <a href="https://shell.azure.com">https://shell.azure.com</a> , or select the <b>Launch Cloud Shell</b> button to open Cloud Shell in your browser.	
Select the <b>Cloud Shell</b> button on the menu bar at the upper right in the <a href="#">Azure portal</a> .	

To run the code in this article in Azure Cloud Shell:

1. Start Cloud Shell.
2. Select the **Copy** button on a code block to copy the code.
3. Paste the code into the Cloud Shell session by selecting **Ctrl+Shift+V** on Windows and Linux or by selecting **Cmd+Shift+V** on macOS.
4. Select **Enter** to run the code.

Terraform is installed by default in the [Cloud Shell](#). If you choose to install Terraform locally, complete the next step, otherwise continue to [Set up Terraform access to Azure](#).

## Install Terraform

To install Terraform, [download](#) the appropriate package for your operating system into a separate install directory. The download contains a single executable file, for which you should also define a global path. For instructions on how to set the path on Linux and Mac, go to [this webpage](#). For instructions on how to set the path on Windows, go to [this webpage](#).

Verify your path configuration with the `terraform` command. A list of available Terraform options is shown, as in the following example output:

```
azureuser@Azure:~$ terraform
Usage: terraform [--version] [--help] <command> [args]
```

## Set up Terraform access to Azure

To enable Terraform to provision resources into Azure, create an [Azure AD service principal](#). The service principal grants your Terraform scripts to provision resources in your Azure subscription.

If you have multiple Azure subscriptions, first query your account with [az account list](#) to get a list of subscription ID and tenant ID values:

```
az account list --query "[].{name:name, subscriptionId:id, tenantId:tenantId}"
```

To use a selected subscription, set the subscription for this session with [az account set](#). Set the `SUBSCRIPTION_ID` environment variable to hold the value of the returned `id` field from the subscription you want to use:

```
az account set --subscription="${SUBSCRIPTION_ID}"
```

Now you can create a service principal for use with Terraform. Use [az ad sp create-for-rbac](#), and set the *scope* to your subscription as follows:

```
az ad sp create-for-rbac --role="Contributor" --scopes="/subscriptions/${SUBSCRIPTION_ID}"
```

Your `appId`, `password`, `sp_name`, and `tenant` are returned. Make a note of the `appId` and `password`.

## Configure Terraform environment variables

To configure Terraform to use your Azure AD service principal, set the following environment variables, which are then used by the [Azure Terraform modules](#). You can also set the environment if working with an Azure cloud other than Azure public.

- `ARM_SUBSCRIPTION_ID`
- `ARM_CLIENT_ID`
- `ARM_CLIENT_SECRET`
- `ARM_TENANT_ID`
- `ARM_ENVIRONMENT`

You can use the following sample shell script to set those variables:

```
#!/bin/sh
echo "Setting environment variables for Terraform"
export ARM_SUBSCRIPTION_ID=your_subscription_id
export ARM_CLIENT_ID=your_appId
export ARM_CLIENT_SECRET=your_password
export ARM_TENANT_ID=your_tenant_id

# Not needed for public, required for usgovernment, german, china
export ARM_ENVIRONMENT=public
```

## Run a sample script

Create a file `test.tf` in an empty directory and paste in the following script.

```
provider "azurerm" {
  # The "feature" block is required for AzureRM provider 2.x.
  # If you are using version 1.x, the "features" block is not allowed.
  version = "~>2.0"
  features {}
}
resource "azurerm_resource_group" "rg" {
  name = "testResourceGroup"
  location = "westus"
}
```

Save the file and then initialize the Terraform deployment. This step downloads the Azure modules required to create an Azure resource group.

```
terraform init
```

The output is similar to the following example:

```
* provider.azurerm: version = "~> 0.3"

Terraform has been successfully initialized!
```

You can preview the actions to be completed by the Terraform script with `terraform plan`. When ready to create the resource group, apply your Terraform plan as follows:

```
terraform apply
```

The output is similar to the following example:

An execution plan has been generated and is shown below.  
Resource actions are indicated with the following symbols:

+ create

Terraform will perform the following actions:

```
+ azurerm_resource_group.rg
  id:          <computed>
  location:    "westus"
  name:        "testResourceGroup"
  tags.%:      <computed>
```

azurerm\_resource\_group.rg: Creating...

```
location: "" => "westus"
name:      "" => "testResourceGroup"
tags.%:    "" => "<computed>"
```

azurerm\_resource\_group.rg: Creation complete after 1s

## Next steps

In this article, you installed Terraform or used the Cloud Shell to configure Azure credentials and start creating resources in your Azure subscription. To create a more complete Terraform deployment in Azure, see the following article:

[Create an Azure VM with Terraform](#)

In this quickstart, you create Azure resources using Terraform. The steps in this article walk you through creating the following resources:

- Azure Cosmos DB instance
- Azure Container Instance
- App that works across these two resources

## Create first configuration

In this section, you create the configuration for an Azure Cosmos DB instance.

1. Sign in to the [Azure portal](#).
2. Open the Azure Cloud Shell.
3. Start the Cloud Shell editor:

```
code main.tf
```

4. The configuration in this step models a couple of Azure resources. These resources include an Azure resource group and an Azure Cosmos DB instance. A random integer is used to create a unique Cosmos DB instance name. Several Cosmos DB settings are also configured. For more information, see the [Cosmos DB Terraform reference](#). Copy and paste the following Terraform configuration into the editor:

```
resource "azurerm_resource_group" "vote-resource-group" {
  name      = "vote-resource-group"
  location  = "westus"
}

resource "random_integer" "ri" {
  min = 10000
  max = 99999
}

resource "azurerm_cosmosdb_account" "vote-cosmos-db" {
  name                = "tfex-cosmos-db-${random_integer.ri.result}"
  location             = azurerm_resource_group.vote-resource-group.location
  resource_group_name = azurerm_resource_group.vote-resource-group.name
  offer_type          = "Standard"
  kind                 = "GlobalDocumentDB"

  consistency_policy {
    consistency_level      = "BoundedStaleness"
    max_interval_in_seconds = 10
    max_staleness_prefix   = 200
  }

  geo_location {
    location      = "westus"
    failover_priority = 0
  }
}
```



5. Save the file (<Ctrl>S) and exit the editor (<Ctrl>Q).

## Run the configuration

In this section, you use several Terraform commands to run the configuration.

1. The `terraform init` command initializes the working directory. Run the following command in Cloud Shell:

```
terraform init
```

2. The `terraform plan` command can be used to validate the configuration syntax. The `-out` parameter directs the results to a file. The output file can be used later to apply the configuration. Run the following command in Cloud Shell:

```
terraform plan --out plan.out
```

3. The `terraform apply` command is used to apply the configuration. The output file from the previous step is specified. This command causes the Azure resources to be created. Run the following command in Cloud Shell:

```
terraform apply plan.out
```

4. To verify the results within the Azure portal, browse to the new resource group. The new Azure Cosmos DB instance is in the new resource group.

## Update configuration

This section shows how to update the configuration to include an Azure Container Instance. The container runs an application that reads and writes data to the Cosmos DB.

1. Start the Cloud Shell editor:

```
code main.tf
```

2. The configuration in this step sets two environment variables: `COSMOS_DB_ENDPOINT` and `COSMOS_DB_MASTERKEY`. These variables hold the location and key for accessing the database. The values for these variables are obtained from the database instance created in the previous step. This process is known as interpolation. To learn more about Terraform interpolation, see [Interpolation Syntax](#). The configuration also includes an output block, which returns the fully qualified domain name (FQDN) of the container instance. Copy and paste the following code into the editor:

```

resource "azurerm_container_group" "vote-aci" {
  name           = "vote-aci"
  location       = azurerm_resource_group.vote-resource-group.location
  resource_group_name = azurerm_resource_group.vote-resource-group.name
  ip_address_type = "public"
  dns_name_label  = "vote-aci"
  os_type         = "linux"

  container {
    name     = "vote-aci"
    image    = "microsoft/azure-vote-front:cosmosdb"
    cpu      = "0.5"
    memory   = "1.5"
    ports {
      port      = 80
      protocol  = "TCP"
    }

    secure_environment_variables = {
      "COSMOS_DB_ENDPOINT" = azurerm_cosmosdb_account.vote-cosmos-db.endpoint
      "COSMOS_DB_MASTERKEY" = azurerm_cosmosdb_account.vote-cosmos-db.primary_master_key
      "TITLE"               = "Azure Voting App"
      "VOTE1VALUE"          = "Cats"
      "VOTE2VALUE"          = "Dogs"
    }
  }
}

output "dns" {
  value = azurerm_container_group.vote-aci.fqdn
}

```

3. Save the file (<Ctrl>S) and exit the editor (<Ctrl>Q).
4. As you did in the previous section, run the following command to visual the changes to be made:

```
terraform plan --out plan.out
```

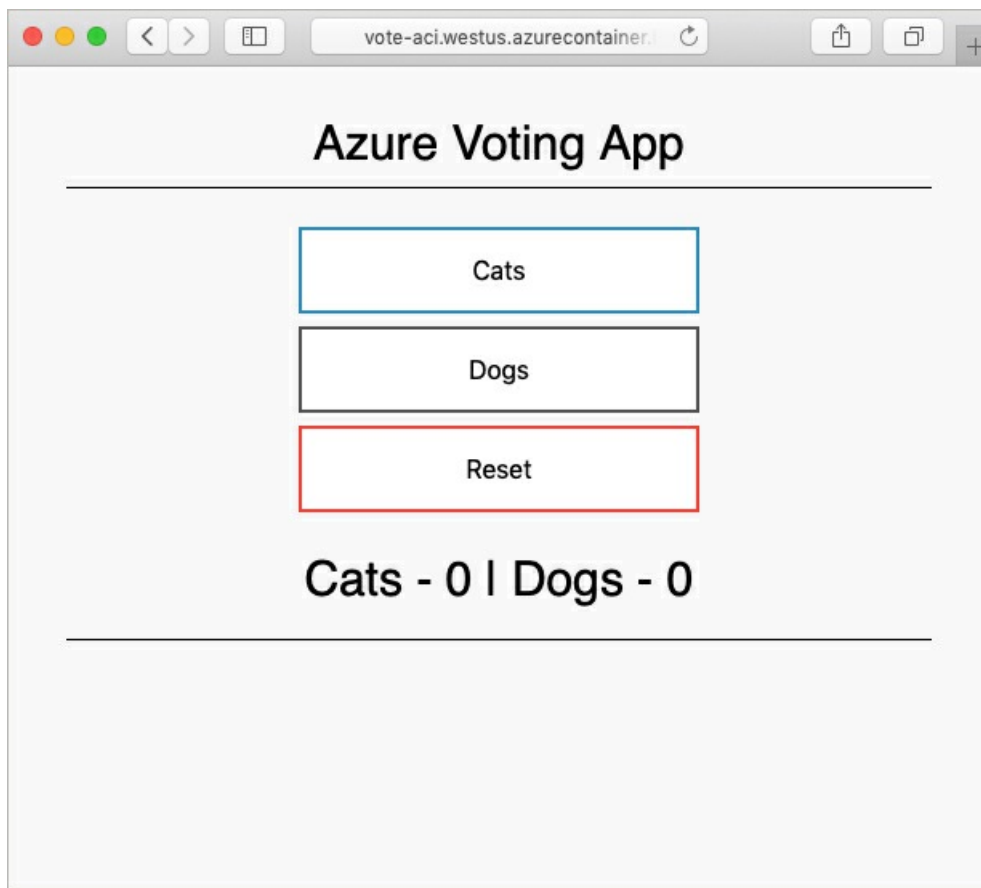
5. Run the `terraform apply` command to apply the configuration.

```
terraform apply plan.out
```

6. Make note of the container instance FQDN.

## Test application

To test the application, navigate to the FQDN of the container instance. You should see results similar to the following output:



## Clean up resources

When no longer needed, delete the resources created in this article.

Run the [terraform destroy](#) command to remove the Azure resources created in this tutorial:

```
terraform destroy -auto-approve
```

## Next steps

[Install and configure Terraform to provision Azure resources.](#)

Terraform allows you to define and create complete infrastructure deployments in Azure. You build Terraform templates in a human-readable format that create and configure Azure resources in a consistent, reproducible manner. This article shows you how to create a complete Linux environment and supporting resources with Terraform. You can also learn how to [install and configure Terraform](#).

#### NOTE

For Terraform specific support, please reach out to Terraform directly using one of their community channels:

- The [Terraform section](#) of the community portal contains questions, use cases, and useful patterns.
- For provider-related questions please visit the [Terraform Providers](#) section of the community portal.

## Create Azure connection and resource group

Let's go through each section of a Terraform template. You can also see the full version of the [Terraform template](#) that you can copy and paste.

The `provider` section tells Terraform to use an Azure provider. To get values for `subscription_id`, `client_id`, `client_secret`, and `tenant_id`, see [Install and configure Terraform](#).

#### TIP

If you create environment variables for the values or are using the [Azure Cloud Shell Bash experience](#), you don't need to include the variable declarations in this section.

```
provider "azurerm" {
  # The "feature" block is required for AzureRM provider 2.x.
  # If you're using version 1.x, the "features" block is not allowed.
  version = "~>2.0"
  features {}

  subscription_id = "xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxxxx"
  client_id       = "xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxxxx"
  client_secret   = "xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxxxx"
  tenant_id      = "xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxxxx"
}
```

The following section creates a resource group named `myResourceGroup` in the `eastus` location:

```
resource "azurerm_resource_group" "myterraformgroup" {
  name     = "myResourceGroup"
  location = "eastus"

  tags = {
    environment = "Terraform Demo"
  }
}
```

In additional sections, you reference the resource group with `${azurerm_resource_group.myterraformgroup.name}`.

## Create virtual network

The following section creates a virtual network named `myVnet` in the `10.0.0.0/16` address space:

```
resource "azurerm_virtual_network" "myterraformnetwork" {
  name                = "myVnet"
  address_space       = ["10.0.0.0/16"]
  location             = "eastus"
  resource_group_name = azurerm_resource_group.myterraformgroup.name

  tags = {
    environment = "Terraform Demo"
  }
}
```

The following section creates a subnet named `mySubnet` in the `myVnet` virtual network:

```
resource "azurerm_subnet" "myterraformsubnet" {
  name                = "mySubnet"
  resource_group_name = azurerm_resource_group.myterraformgroup.name
  virtual_network_name = azurerm_virtual_network.myterraformnetwork.name
  address_prefix       = "10.0.2.0/24"
}
```

## Create public IP address

To access resources across the Internet, create and assign a public IP address to your VM. The following section creates a public IP address named `myPublicIP`:

```
resource "azurerm_public_ip" "myterraformpublicip" {
  name                = "myPublicIP"
  location             = "eastus"
  resource_group_name = azurerm_resource_group.myterraformgroup.name
  allocation_method    = "Dynamic"

  tags = {
    environment = "Terraform Demo"
  }
}
```

## Create Network Security Group

Network Security Groups control the flow of network traffic in and out of your VM. The following section creates a network security group named `myNetworkSecurityGroup` and defines a rule to allow SSH traffic on TCP port 22:

```

resource "azurerm_network_security_group" "myterraformnsg" {
  name           = "myNetworkSecurityGroup"
  location       = "eastus"
  resource_group_name = azurerm_resource_group.myterraformgroup.name

  security_rule {
    name           = "SSH"
    priority       = 1001
    direction      = "Inbound"
    access         = "Allow"
    protocol       = "Tcp"
    source_port_range = "*"
    destination_port_range = "22"
    source_address_prefix = "*"
    destination_address_prefix = "*"
  }

  tags = {
    environment = "Terraform Demo"
  }
}

```

## Create virtual network interface card

A virtual network interface card (NIC) connects your VM to a given virtual network, public IP address, and network security group. The following section in a Terraform template creates a virtual NIC named `myNIC` connected to the virtual networking resources you've created:

```

resource "azurerm_network_interface" "myterraformnic" {
  name           = "myNIC"
  location       = "eastus"
  resource_group_name = azurerm_resource_group.myterraformgroup.name

  ip_configuration {
    name                 = "myNicConfiguration"
    subnet_id            = "${azurerm_subnet.myterraformsubnet.id}"
    private_ip_address_allocation = "Dynamic"
    public_ip_address_id = "${azurerm_public_ip.myterraformpublicip.id}"
  }

  tags = {
    environment = "Terraform Demo"
  }
}

# Connect the security group to the network interface
resource "azurerm_network_interface_security_group_association" "example" {
  network_interface_id      = azurerm_network_interface.myterraformnic.id
  network_security_group_id = azurerm_network_security_group.myterraformnsg.id
}

```

## Create storage account for diagnostics

To store boot diagnostics for a VM, you need a storage account. These boot diagnostics can help you troubleshoot problems and monitor the status of your VM. The storage account you create is only to store the boot diagnostics data. As each storage account must have a unique name, the following section generates some random text:

```
resource "random_id" "randomId" {
  keepers = {
    # Generate a new ID only when a new resource group is defined
    resource_group = azurerm_resource_group.myterraformgroup.name
  }

  byte_length = 8
}
```

Now you can create a storage account. The following section creates a storage account, with the name based on the random text generated in the preceding step:

```
resource "azurerm_storage_account" "mystorageaccount" {
  name                        = "diag${random_id.randomId.hex}"
  resource_group_name        = azurerm_resource_group.myterraformgroup.name
  location                   = "eastus"
  account_replication_type   = "LRS"
  account_tier                = "Standard"

  tags = {
    environment = "Terraform Demo"
  }
}
```

## Create virtual machine

The final step is to create a VM and use all the resources created. The following section creates a VM named `myVM` and attaches the virtual NIC named `myNIC`. The latest `Ubuntu 16.04-LTS` image is used, and a user named `azureuser` is created with password authentication disabled.

SSH key data is provided in the `ssh_keys` section. Provide a public SSH key in the `key_data` field.

```

resource "azurerm_linux_virtual_machine" "myterraformvm" {
  name                = "myVM"
  location            = "eastus"
  resource_group_name = azurerm_resource_group.myterraformgroup.name
  network_interface_ids = [azurerm_network_interface.myterraformnic.id]
  size                = "Standard_DS1_v2"

  os_disk {
    name                = "myOsDisk"
    caching             = "ReadWrite"
    storage_account_type = "Premium_LRS"
  }

  source_image_reference {
    publisher = "Canonical"
    offer     = "UbuntuServer"
    sku       = "16.04.0-LTS"
    version   = "latest"
  }

  computer_name = "myvm"
  admin_username = "azureuser"
  disable_password_authentication = true

  admin_ssh_key {
    username   = "azureuser"
    public_key = file("/home/azureuser/.ssh/authorized_keys")
  }

  boot_diagnostics {
    storage_account_uri = azurerm_storage_account.mystorageaccount.primary_blob_endpoint
  }

  tags = {
    environment = "Terraform Demo"
  }
}

```

## Complete Terraform script

To bring all these sections together and see Terraform in action, create a file called `terraform_azure.tf` and paste the following content:

```

# Configure the Microsoft Azure Provider
provider "azurerm" {
  # The "feature" block is required for AzureRM provider 2.x.
  # If you're using version 1.x, the "features" block is not allowed.
  version = "~>2.0"
  features {}

  subscription_id = "xxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxxx"
  client_id       = "xxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxxx"
  client_secret   = "xxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxxx"
  tenant_id       = "xxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxxx"
}

# Create a resource group if it doesn't exist
resource "azurerm_resource_group" "myterraformgroup" {
  name     = "myResourceGroup"
  location = "eastus"

  tags = {
    environment = "Terraform Demo"
  }
}

```



```

# Create virtual network
resource "azurerm_virtual_network" "myterraformnetwork" {
  name                = "myVnet"
  address_space       = ["10.0.0.0/16"]
  location            = "eastus"
  resource_group_name = azurerm_resource_group.myterraformgroup.name

  tags = {
    environment = "Terraform Demo"
  }
}

# Create subnet
resource "azurerm_subnet" "myterraformsubnet" {
  name                = "mySubnet"
  resource_group_name = azurerm_resource_group.myterraformgroup.name
  virtual_network_name = azurerm_virtual_network.myterraformnetwork.name
  address_prefix       = "10.0.1.0/24"
}

# Create public IPs
resource "azurerm_public_ip" "myterraformpublicip" {
  name                = "myPublicIP"
  location            = "eastus"
  resource_group_name = azurerm_resource_group.myterraformgroup.name
  allocation_method   = "Dynamic"

  tags = {
    environment = "Terraform Demo"
  }
}

# Create Network Security Group and rule
resource "azurerm_network_security_group" "myterraformnsg" {
  name                = "myNetworkSecurityGroup"
  location            = "eastus"
  resource_group_name = azurerm_resource_group.myterraformgroup.name

  security_rule {
    name                = "SSH"
    priority            = 1001
    direction           = "Inbound"
    access              = "Allow"
    protocol            = "Tcp"
    source_port_range   = "*"
    destination_port_range = "22"
    source_address_prefix = "*"
    destination_address_prefix = "*"
  }

  tags = {
    environment = "Terraform Demo"
  }
}

# Create network interface
resource "azurerm_network_interface" "myterraformnic" {
  name                = "myNIC"
  location            = "eastus"
  resource_group_name = azurerm_resource_group.myterraformgroup.name

  ip_configuration {
    name                = "myNicConfiguration"
    subnet_id           = azurerm_subnet.myterraformsubnet.id
    private_ip_address_allocation = "Dynamic"
    public_ip_address_id = azurerm_public_ip.myterraformpublicip.id
  }
}

```

```

    tags = {
        environment = "Terraform Demo"
    }
}

# Connect the security group to the network interface
resource "azurerm_network_interface_security_group_association" "example" {
    network_interface_id      = azurerm_network_interface.myterraformnic.id
    network_security_group_id = azurerm_network_security_group.myterraformmsg.id
}

# Generate random text for a unique storage account name
resource "random_id" "randomId" {
    keepers = {
        # Generate a new ID only when a new resource group is defined
        resource_group = azurerm_resource_group.myterraformgroup.name
    }

    byte_length = 8
}

# Create storage account for boot diagnostics
resource "azurerm_storage_account" "mystorageaccount" {
    name                = "diag${random_id.randomId.hex}"
    resource_group_name = azurerm_resource_group.myterraformgroup.name
    location            = "eastus"
    account_tier        = "Standard"
    account_replication_type = "LRS"

    tags = {
        environment = "Terraform Demo"
    }
}

# Create virtual machine
resource "azurerm_linux_virtual_machine" "myterraformvm" {
    name                = "myVM"
    location            = "eastus"
    resource_group_name = azurerm_resource_group.myterraformgroup.name
    network_interface_ids = [azurerm_network_interface.myterraformnic.id]
    size                = "Standard_DS1_v2"

    os_disk {
        name                = "myOsDisk"
        caching             = "ReadWrite"
        storage_account_type = "Premium_LRS"
    }

    source_image_reference {
        publisher = "Canonical"
        offer     = "UbuntuServer"
        sku       = "16.04.0-LTS"
        version   = "latest"
    }

    computer_name = "myvm"
    admin_username = "azureuser"
    disable_password_authentication = true

    admin_ssh_key {
        username   = "azureuser"
        public_key = file("/home/azureuser/.ssh/authorized_keys")
    }

    boot_diagnostics {
        storage_account_uri = azurerm_storage_account.mystorageaccount.primary_blob_endpoint
    }
}

```

```
tags = {  
  environment = "Terraform Demo"  
}
```

## Build and deploy the infrastructure

With your Terraform template created, the first step is to initialize Terraform. This step ensures that Terraform has all the prerequisites to build your template in Azure.

```
terraform init
```

The next step is to have Terraform review and validate the template. This step compares the requested resources to the state information saved by Terraform and then outputs the planned execution. The Azure resources aren't created at this point.

```
terraform plan
```

After you execute the previous command, you should see something like the following screen:

```
Refreshing Terraform state in-memory prior to plan...  
The refreshed state will be used to calculate this plan, but will not be  
persisted to local or remote state storage.  
  
...  
  
Note: You didn't specify an "-out" parameter to save this plan, so when  
"apply" is called, Terraform can't guarantee this is what will execute.  
+ azurerm_resource_group.myterraform  
  <snip>  
+ azurerm_virtual_network.myterraformnetwork  
  <snip>  
+ azurerm_network_interface.myterraformnic  
  <snip>  
+ azurerm_network_security_group.myterraformnsg  
  <snip>  
+ azurerm_public_ip.myterraformpublicip  
  <snip>  
+ azurerm_subnet.myterraformsubnet  
  <snip>  
+ azurerm_virtual_machine.myterraformvm  
  <snip>  
Plan: 7 to add, 0 to change, 0 to destroy.
```

If everything looks correct and you're ready to build the infrastructure in Azure, apply the template in Terraform:

```
terraform apply
```

Once Terraform completes, your VM infrastructure is ready. Obtain the public IP address of your VM with [az vm show](#):

```
az vm show --resource-group myResourceGroup --name myVM -d --query [publicIps] --o tsv
```

You can then SSH to your VM:

```
ssh azureuser@<publicIp>
```

## Next steps

[Learn more about using Terraform in Azure](#)

# Tutorial: Store Terraform state in Azure Storage

1/7/2020 • 3 minutes to read • [Edit Online](#)

Terraform state is used to reconcile deployed resources with Terraform configurations. State allows Terraform to know what Azure resources to add, update, or delete. By default, Terraform state is stored locally when you run the `terraform apply` command. This configuration isn't ideal for the following reasons:

- Local state doesn't work well in a team or collaborative environment.
- Terraform state can include sensitive information.
- Storing state locally increases the chance of inadvertent deletion.

Terraform supports the persisting of state in remote storage. One such supported back end is Azure Storage. This document shows how to configure and use Azure Storage for this purpose.

## Configure storage account

Before you use Azure Storage as a back end, you must create a storage account. The storage account can be created with the Azure portal, PowerShell, the Azure CLI, or Terraform itself. Use the following sample to configure the storage account with the Azure CLI.

```
#!/bin/bash

RESOURCE_GROUP_NAME=tstate
STORAGE_ACCOUNT_NAME=tstate$RANDOM
CONTAINER_NAME=tstate

# Create resource group
az group create --name $RESOURCE_GROUP_NAME --location eastus

# Create storage account
az storage account create --resource-group $RESOURCE_GROUP_NAME --name $STORAGE_ACCOUNT_NAME --sku Standard_LRS --encryption-services blob

# Get storage account key
ACCOUNT_KEY=$(az storage account keys list --resource-group $RESOURCE_GROUP_NAME --account-name $STORAGE_ACCOUNT_NAME --query [0].value -o tsv)

# Create blob container
az storage container create --name $CONTAINER_NAME --account-name $STORAGE_ACCOUNT_NAME --account-key $ACCOUNT_KEY

echo "storage_account_name: $STORAGE_ACCOUNT_NAME"
echo "container_name: $CONTAINER_NAME"
echo "access_key: $ACCOUNT_KEY"
```

Take note of the storage account name, container name, and storage access key. These values are needed when you configure the remote state.

## Configure state back end

The Terraform state back end is configured when you run the `terraform init` command. The following data is needed to configure the state back end:

- **storage\_account\_name:** The name of the Azure Storage account.
- **container\_name:** The name of the blob container.

- **key**: The name of the state store file to be created.
- **access\_key**: The storage access key.

Each of these values can be specified in the Terraform configuration file or on the command line. We recommend that you use an environment variable for the `access_key` value. Using an environment variable prevents the key from being written to disk.

Create an environment variable named `ARM_ACCESS_KEY` with the value of the Azure Storage access key.

```
export ARM_ACCESS_KEY=<storage access key>
```

To further protect the Azure Storage account access key, store it in Azure Key Vault. The environment variable can then be set by using a command similar to the following. For more information on Azure Key Vault, see the [Azure Key Vault documentation](#).

```
export ARM_ACCESS_KEY=$(az keyvault secret show --name terraform-backend-key --vault-name myKeyVault --query value -o tsv)
```

To configure Terraform to use the back end, the following steps need to be done:

- Include a `backend` configuration block with a type of `azurerm`.
- Add a `storage_account_name` value to the configuration block.
- Add a `container_name` value to the configuration block.
- Add a `key` value to the configuration block.

The following example configures a Terraform back end and creates an Azure resource group.

```
terraform {
  backend "azurerm" {
    resource_group_name = "tstate"
    storage_account_name = "tstate09762"
    container_name      = "tstate"
    key                 = "terraform.tfstate"
  }
}

resource "azurerm_resource_group" "state-demo-secure" {
  name     = "state-demo"
  location = "eastus"
}
```

Initialize the configuration by doing the following steps:

1. Run the `terraform init` command.
2. Run the `terraform apply` command.

You can now find the state file in the Azure Storage blob.

## State locking

Azure Storage blobs are automatically locked before any operation that writes state. This pattern prevents concurrent state operations, which can cause corruption.

For more information, see [State locking](#) in the Terraform documentation.

You can see the lock when you examine the blob through the Azure portal or other Azure management tooling.

terraform.tfstate

Blob

Save

Discard

Refresh

Download

Acquire lease

Break lease

Overview

Snapshots

Edit blob

Generate SAS

Properties

URL

https://tfstatestorage2480.blob.core.windo...

LAST MODIFIED

9/13/2018, 2:07:56 PM

CREATION TIME

9/13/2018, 2:07:56 PM

TYPE

Block blob

SIZE

282 B

SERVER ENCRYPTED

true

ETAG

0x8D619BCFA76B606

CONTENT-MD5

-

LEASE STATUS

Locked

LEASE STATE

Leased

LEASE DURATION

Infinite

COPY STATUS

-

COPY COMPLETION TIME

-

Undelete all snapshots

## Encryption at rest

Data stored in an Azure blob is encrypted before being persisted. When needed, Terraform retrieves the state from the back end and stores it in local memory. Using this pattern, state is never written to your local disk.

For more information on Azure Storage encryption, see [Azure Storage service encryption for data at rest](#).

## Next steps




[Learn more about using Terraform in Azure](#)

Terraform works well from a Bash command line in macOS, Windows, or Linux. Running your Terraform configurations in the Bash experience of [Azure Cloud Shell](#) has some unique advantages. This tutorial shows how to write Terraform scripts that deploy to Azure by using Cloud Shell.

## Use Azure Cloud Shell

Azure hosts Azure Cloud Shell, an interactive shell environment that you can use through your browser. You can use either Bash or PowerShell with Cloud Shell to work with Azure services. You can use the Cloud Shell preinstalled commands to run the code in this article without having to install anything on your local environment.

To start Azure Cloud Shell:

OPTION	EXAMPLE/LINK
Select <b>Try It</b> in the upper-right corner of a code block. Selecting <b>Try It</b> doesn't automatically copy the code to Cloud Shell.	
Go to <a href="https://shell.azure.com">https://shell.azure.com</a> , or select the <b>Launch Cloud Shell</b> button to open Cloud Shell in your browser.	
Select the <b>Cloud Shell</b> button on the menu bar at the upper right in the <a href="#">Azure portal</a> .	

To run the code in this article in Azure Cloud Shell:

1. Start Cloud Shell.
2. Select the **Copy** button on a code block to copy the code.
3. Paste the code into the Cloud Shell session by selecting **Ctrl+Shift+V** on Windows and Linux or by selecting **Cmd+Shift+V** on macOS.
4. Select **Enter** to run the code.

## Automatic credential configuration

Terraform is installed and immediately available in Cloud Shell. Terraform scripts authenticate with Azure when logged in to Cloud Shell to manage infrastructure without any additional configuration. Automatic authentication bypasses two manual processes:

- Creating an Active Directory service principal
- Configuring the Azure Terraform provider variables

## Use modules and providers

Azure Terraform modules require credentials to access and modify Azure resources. To use Terraform modules in Cloud Shell, add the following code:



```
# Configure the Microsoft Azure Provider
provider "azurerm" {
  # The "feature" block is required for AzureRM provider 2.x.
  # If you are using version 1.x, the "features" block is not allowed.
  version = "~>2.0"
  features {}
}
```

Cloud Shell passes required values for the `azurerm` provider through environment variables when you use any of the `terraform` CLI commands.

## Other Cloud Shell developer tools

Files and shell states persist in Azure Storage between Cloud Shell sessions. Use [Azure Storage Explorer](#) to copy and upload files to Cloud Shell from your local computer.

The Azure CLI is available in Cloud Shell and is a great tool for testing configurations and checking your work after `terraform apply` or `terraform destroy` finishes.

## Next steps

[Create a small VM cluster using the Module Registry](#)

You can use [Azure deployment slots](#) to swap between different versions of your app. That ability helps you minimize the impact of broken deployments.

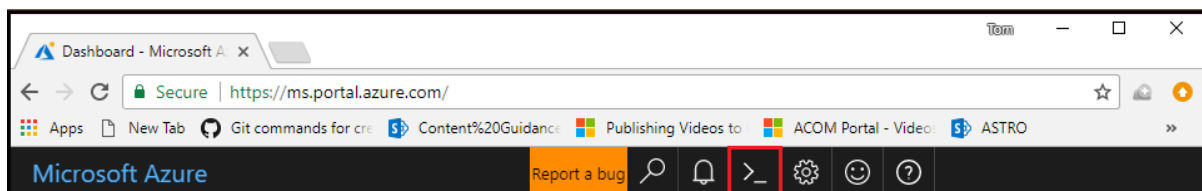
This article illustrates an example use of deployment slots by walking you through the deployment of two apps via GitHub and Azure. One app is hosted in a production slot. The second app is hosted in a staging slot. (The names "production" and "staging" are arbitrary. They can be whatever is appropriate for your scenario.) After you configure your deployment slots, you use Terraform to swap between the two slots as needed.

## Prerequisites

- **Azure subscription:** If you don't have an Azure subscription, create a [free account](#) before you begin.
- **GitHub account:** You need a [GitHub](#) account to fork and use the test GitHub repo.

## Create and apply the Terraform plan

1. Browse to the [Azure portal](#).
2. Open [Azure Cloud Shell](#). If you didn't select an environment previously, select **Bash** as your environment.



3. Change directories to the `clouddrive` directory.

```
cd clouddrive
```

4. Create a directory named `deploy`.

```
mkdir deploy
```

5. Create a directory named `swap`.

```
mkdir swap
```

6. Use the `ls` bash command to verify that you successfully created both directories.

```
Bash
Requesting a Cloud Shell.Succeeded.
Connecting terminal...

tom@Azure:~$ cd clouddrive
tom@Azure:~/clouddrive$ mkdir deploy
tom@Azure:~/clouddrive$ mkdir swap
tom@Azure:~/clouddrive$ ls
deploy  swap
tom@Azure:~/clouddrive$
```

7. Change directories to the `deploy` directory.

```
cd deploy
```

8. In Cloud Shell, create a file named `deploy.tf`.

```
code deploy.tf
```

9. Paste the following code into the editor:

```
# Configure the Azure provider
provider "azurerm" {
  # The "feature" block is required for AzureRM provider 2.x.
  # If you are using version 1.x, the "features" block is not allowed.
  version = "~>2.0"
  features {}
}

resource "azurerm_resource_group" "slotDemo" {
  name     = "slotDemoResourceGroup"
  location = "westus2"
}

resource "azurerm_app_service_plan" "slotDemo" {
  name                = "slotAppServicePlan"
  location             = azurerm_resource_group.slotDemo.location
  resource_group_name = azurerm_resource_group.slotDemo.name
  sku {
    tier = "Standard"
    size = "S1"
  }
}

resource "azurerm_app_service" "slotDemo" {
  name                = "slotAppService"
  location             = azurerm_resource_group.slotDemo.location
  resource_group_name = azurerm_resource_group.slotDemo.name
  app_service_plan_id = azurerm_app_service_plan.slotDemo.id
}

resource "azurerm_app_service_slot" "slotDemo" {
  name                = "slotAppServiceSlotOne"
  location             = azurerm_resource_group.slotDemo.location
  resource_group_name = azurerm_resource_group.slotDemo.name
  app_service_plan_id = azurerm_app_service_plan.slotDemo.id
  app_service_name     = azurerm_app_service.slotDemo.name
}
```

10. Save the file (`<Ctrl>S`) and exit the editor (`<Ctrl>Q`).

11. Now that you've created the file, verify its contents.

```
cat deploy.tf
```

12. Initialize Terraform.

```
terraform init
```

13. Create the Terraform plan.

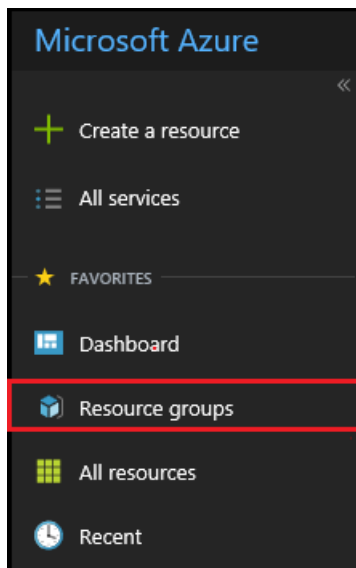
```
terraform plan
```

14. Provision the resources that are defined in the `deploy.tf` configuration file. (Confirm the action by entering `yes` at the prompt.)

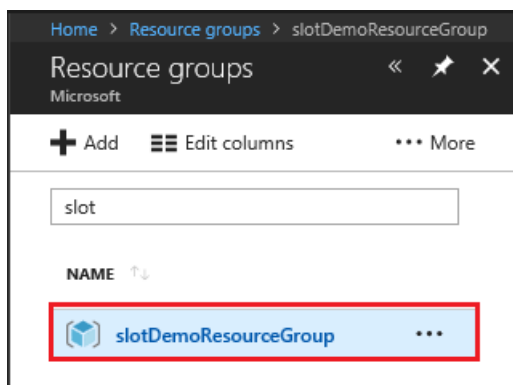
```
terraform apply
```

15. Close the Cloud Shell window.

16. On the main menu of the Azure portal, select **Resource groups**.



17. On the **Resource groups** tab, select `slotDemoResourceGroup`.



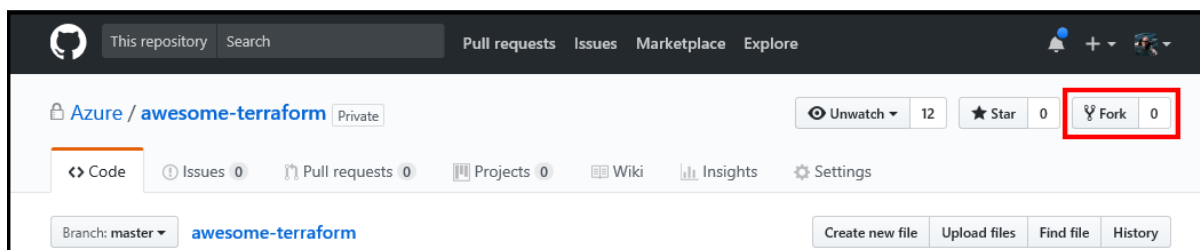
You now see all the resources that Terraform has created.

Filter by name...	All types	All locations
3 items	<input checked="" type="checkbox"/> Show hidden types ⓘ	
<input type="checkbox"/> NAME ↑↓		TYPE ↑↓
<input type="checkbox"/> slotAppService		App Service
<input type="checkbox"/> slotAppServiceSlotOne (slotAppService/slotAppServiceSlotOne)		Web App
<input type="checkbox"/> slotAppServicePlan		App Service plan

## Fork the test project

Before you can test the creation and swapping in and out of the deployment slots, you need to fork the test project from GitHub.

1. Browse to the [awesome-terraform repo on GitHub](#).
2. Fork the **awesome-terraform** repo.

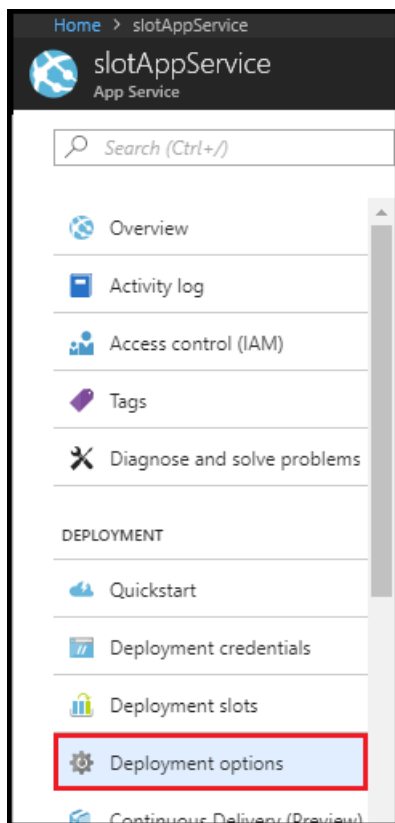


3. Follow any prompts to fork to your environment.

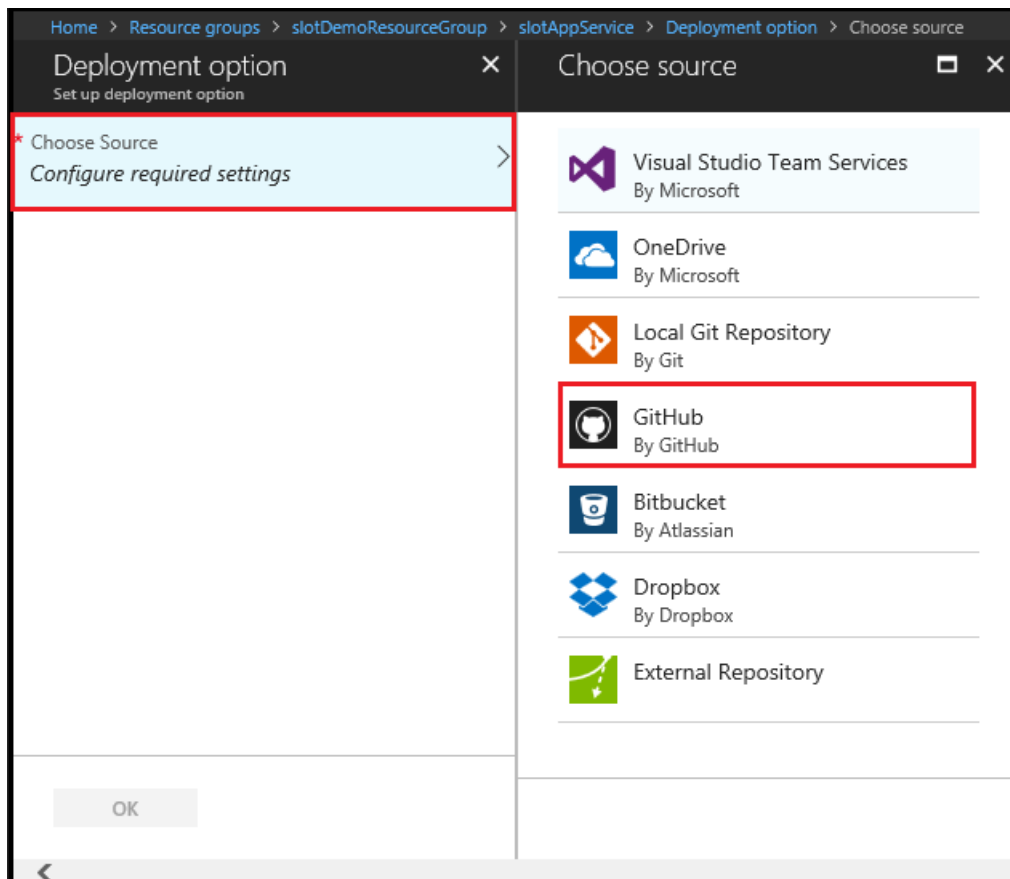
## Deploy from GitHub to your deployment slots

After you fork the test project repo, configure the deployment slots via the following steps:

1. On the main menu of the Azure portal, select **Resource groups**.
2. Select **slotDemoResourceGroup**.
3. Select **slotAppService**.
4. Select **Deployment options**.

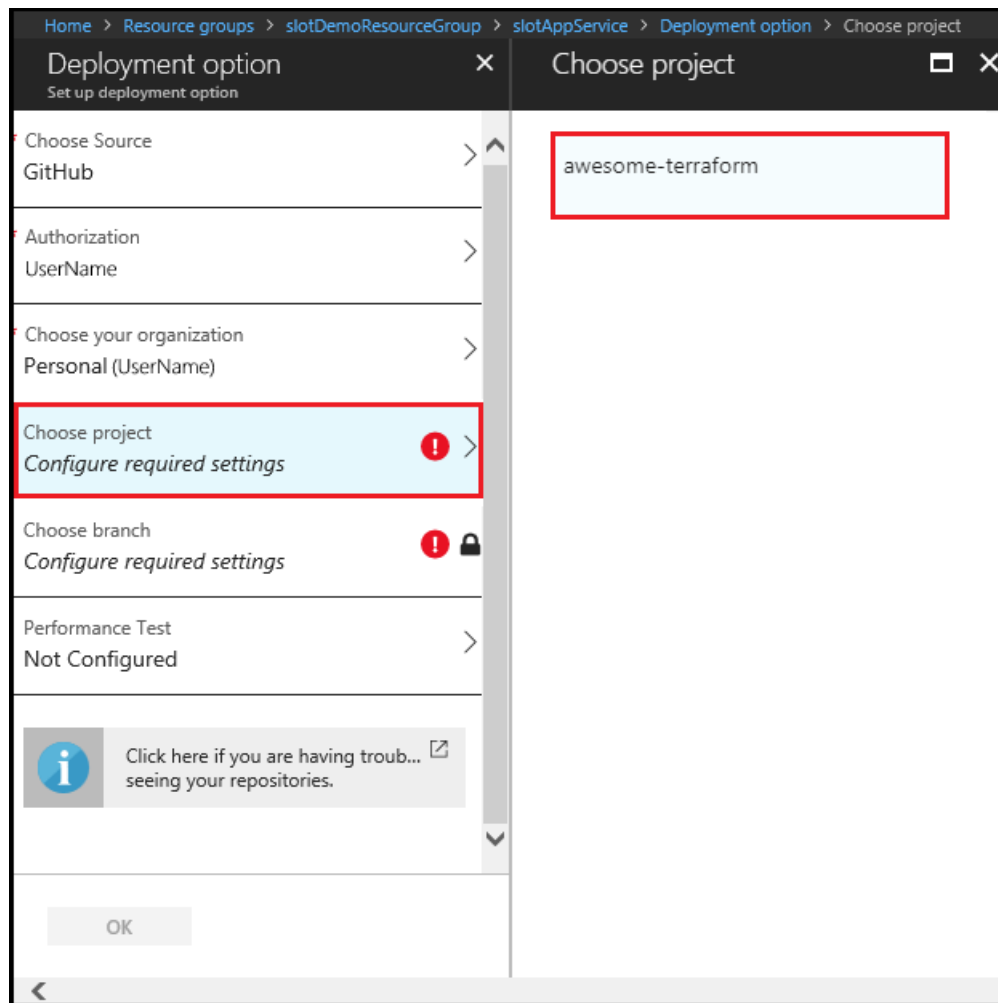


5. On the **Deployment option** tab, select **Choose Source**, and then select **GitHub**.

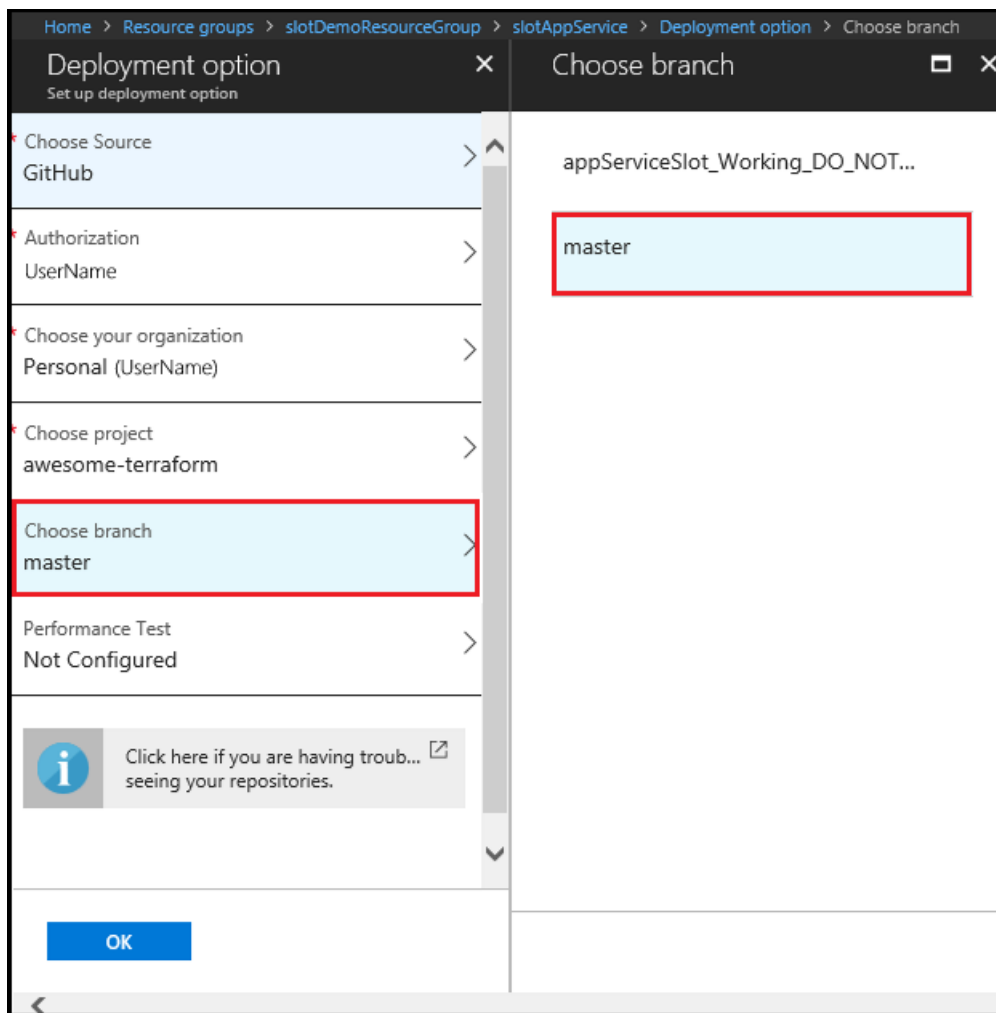


6. After Azure makes the connection and displays all the options, select **Authorization**.
7. On the **Authorization** tab, select **Authorize**, and supply the credentials that Azure needs to access your GitHub account.
8. After Azure validates your GitHub credentials, a message appears and says that the authorization process has finished. Select **OK** to close the **Authorization** tab.

9. Select **Choose your organization** and select your organization.
10. Select **Choose project**.
11. On the **Choose project** tab, select the **awesome-terraform** project.



12. Select **Choose branch**.
13. On the **Choose branch** tab, select **master**.

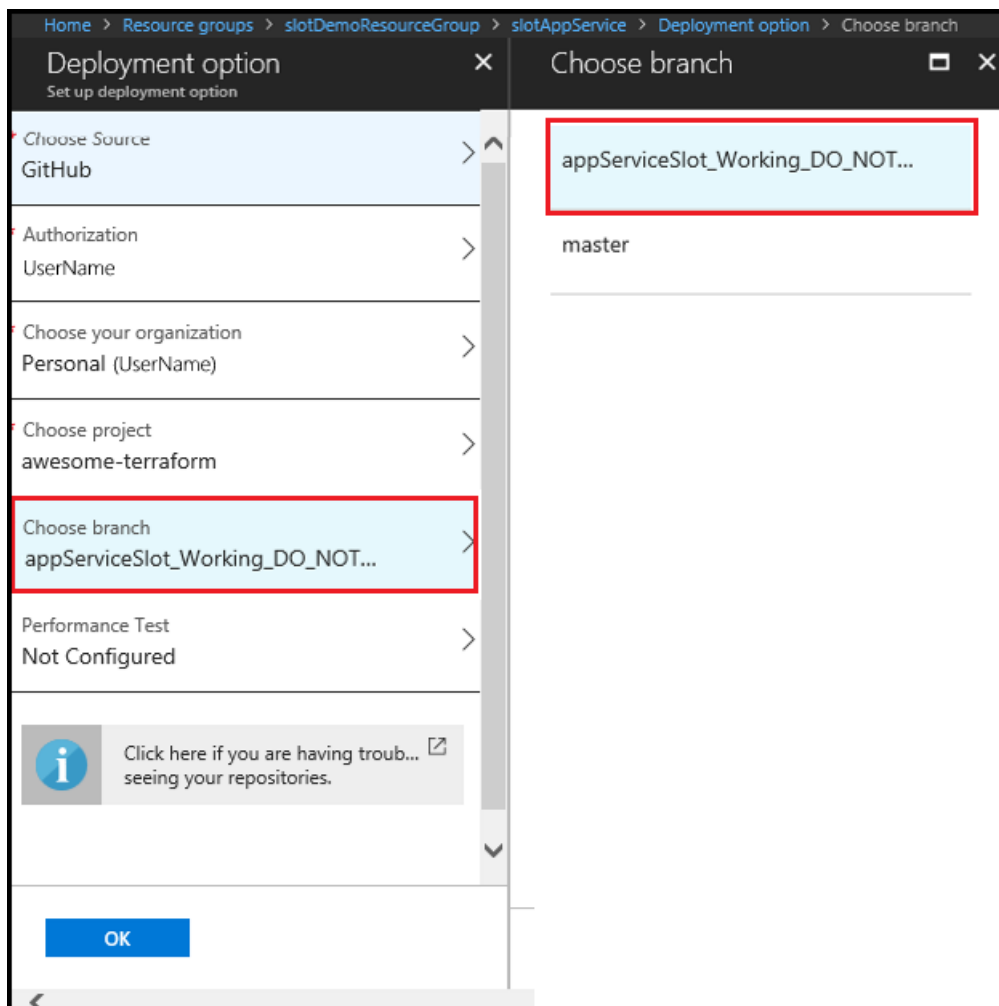


14. On the **Deployment option** tab, select OK.

At this point, you've deployed the production slot. To deploy the staging slot, do the previous steps with the following modifications:

- In step 3, select the **slotAppServiceSlotOne** resource.
- In step 13, select the working branch instead of the master branch.

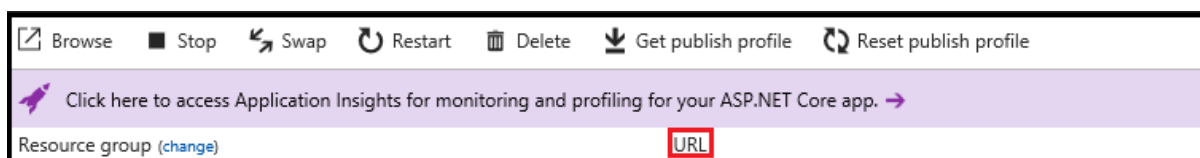




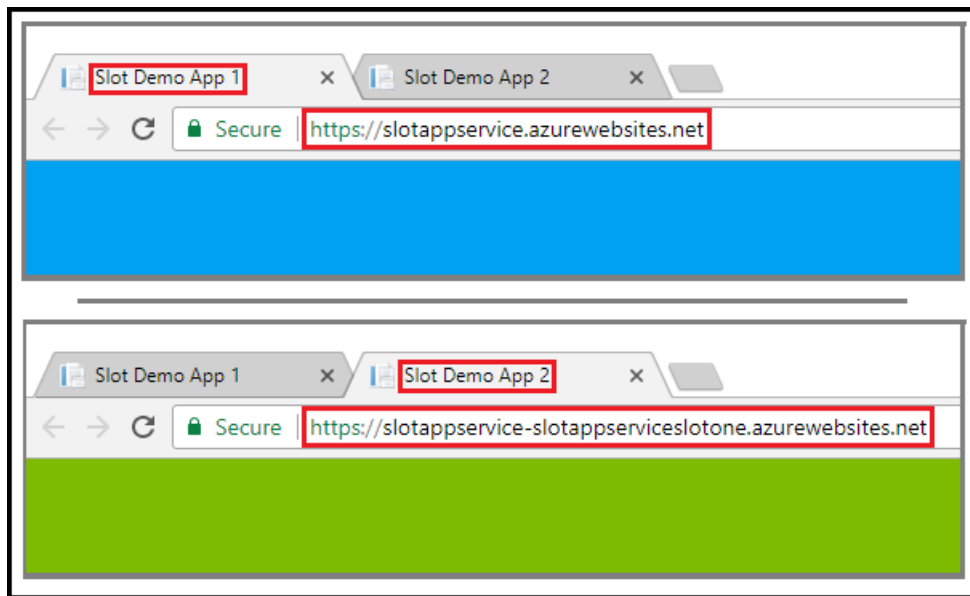
## Test the app deployments

In the previous sections, you set up two slots--`slotAppService` and `slotAppServiceSlotOne`--to deploy from different branches in GitHub. Let's preview the web apps to validate that they were successfully deployed.

1. On the main menu of the Azure portal, select **Resource groups**.
2. Select `slotDemoResourceGroup`.
3. Select either `slotAppService` or `slotAppServiceSlotOne`.
4. On the overview page, select **URL**.



5. Depending on the selected app, you see the following results:
  - `slotAppService` web app - Blue page with a page title of **Slot Demo App 1**.
  - `slotAppServiceSlotOne` web app - Green page with a page title of **Slot Demo App 2**.



## Swap the two deployment slots

To test swapping the two deployment slots, do the following steps:

1. Switch to the browser tab that's running **slotAppService** (the app with the blue page).
2. Return to the Azure portal on a separate tab.
3. Open Cloud Shell.
4. Change directories to the **clouddrive/swap** directory.

```
cd clouddrive/swap
```

5. In Cloud Shell, create a file named `swap.tf`.

```
code swap.tf
```

6. Paste the following code into the editor:

```
# Configure the Azure provider
provider "azurerm" {
  # The "feature" block is required for AzureRM provider 2.x.
  # If you are using version 1.x, the "features" block is not allowed.
  version = "~>2.0"
  features {}
}

# Swap the production slot and the staging slot
resource "azurerm_app_service_active_slot" "slotDemoActiveSlot" {
  resource_group_name = "slotDemoResourceGroup"
  app_service_name    = "slotAppService"
  app_service_slot_name = "slotAppServiceSlotOne"
}
```

7. Save the file (**<Ctrl>S**) and exit the editor (**<Ctrl>Q**).
8. Initialize Terraform.

```
terraform init
```

9. Create the Terraform plan.

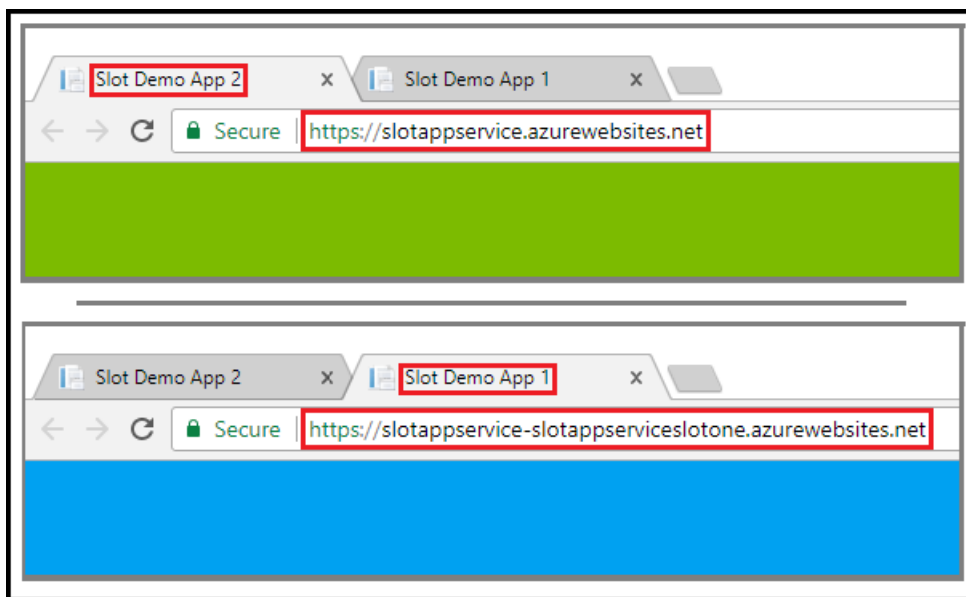
```
terraform plan
```

10. Provision the resources that are defined in the `swap.tf` configuration file. (Confirm the action by entering `yes` at the prompt.)

```
terraform apply
```

11. After Terraform has swapped the slots, return to the browser. Refresh the page.

The web app in your **slotAppServiceSlotOne** staging slot has been swapped with the production slot and is now rendered in green.



To return to the original production version of the app, reapply the Terraform plan that you created from the `swap.tf` configuration file.

```
terraform apply
```

After the app is swapped, you see the original configuration.

## Next steps

[Learn more about using Terraform in Azure](#)

# Tutorial: Create a Kubernetes cluster with Azure Kubernetes Service using Terraform

4/2/2020 • 7 minutes to read • [Edit Online](#)

[Azure Kubernetes Service \(AKS\)](#) manages your hosted Kubernetes environment. AKS allows you to deploy and manage containerized applications without container orchestration expertise. AKS also enables you to do many common maintenance operations without taking your app offline. These operations include provisioning, upgrading, and scaling resources on demand.

In this tutorial, you learn how to do the following tasks:

- Use HCL (HashiCorp Language) to define a Kubernetes cluster
- Use Terraform and AKS to create a Kubernetes cluster
- Use the kubectl tool to test the availability of a Kubernetes cluster

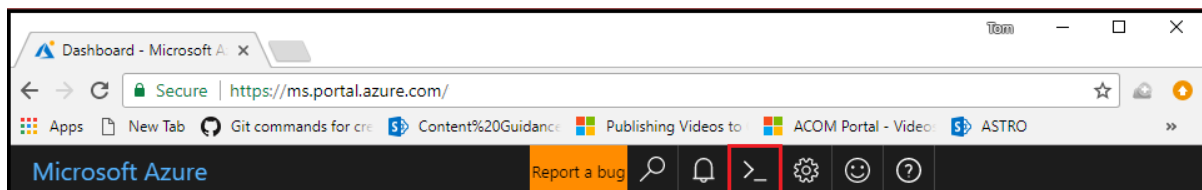
## Prerequisites

- **Azure subscription:** If you don't have an Azure subscription, create a [free account](#) before you begin.
- **Configure Terraform:** Follow the directions in the article, [Terraform and configure access to Azure](#)
- **Azure service principal:** Follow the directions in the **Create the service principal** section in the article, [Create an Azure service principal with Azure CLI](#). Take note of the values for the appld, displayName, password, and tenant.

## Create the directory structure

The first step is to create the directory that holds your Terraform configuration files for the exercise.

1. Browse to the [Azure portal](#).
2. Open [Azure Cloud Shell](#). If you didn't select an environment previously, select **Bash** as your environment.



3. Change directories to the `clouddrive` directory.

```
cd clouddrive
```

4. Create a directory named `terraform-aks-k8s`.

```
mkdir terraform-aks-k8s
```

5. Change directories to the new directory:

```
cd terraform-aks-k8s
```

# Declare the Azure provider

Create the Terraform configuration file that declares the Azure provider.

1. In Cloud Shell, create a file named `main.tf`.

```
code main.tf
```

2. Paste the following code into the editor:

```
provider "azurerm" {  
    # The "feature" block is required for AzureRM provider 2.x.  
    # If you are using version 1.x, the "features" block is not allowed.  
    version = "~>2.0"  
    features {}  
}  
  
terraform {  
    backend "azurerm" {}  
}
```

3. Save the file (<Ctrl>S) and exit the editor (<Ctrl>Q).

# Define a Kubernetes cluster

Create the Terraform configuration file that declares the resources for the Kubernetes cluster.

1. In Cloud Shell, create a file named `k8s.tf`.

```
code k8s.tf
```

2. Paste the following code into the editor:

```
resource "azurerm_resource_group" "k8s" {  
    name      = var.resource_group_name  
    location  = var.location  
}  
  
resource "random_id" "log_analytics_workspace_name_suffix" {  
    byte_length = 8  
}  
  
resource "azurerm_log_analytics_workspace" "test" {  
    # The Workspace name has to be unique across the whole of azure, not just the current  
    # subscription/tenant.  
    name = "${var.log_analytics_workspace_name}-${  
    ${random_id.log_analytics_workspace_name_suffix.dec}}"  
    location          = var.log_analytics_workspace_location  
    resource_group_name = azurerm_resource_group.k8s.name  
    sku               = var.log_analytics_workspace_sku  
}  
  
resource "azurerm_log_analytics_solution" "test" {  
    solution_name      = "ContainerInsights"  
    location           = azurerm_log_analytics_workspace.test.location  
    resource_group_name = azurerm_resource_group.k8s.name  
    workspace_resource_id = azurerm_log_analytics_workspace.test.id  
    workspace_name      = azurerm_log_analytics_workspace.test.name  
  
    plan {  
        publisher = "Microsoft"
```

```

        product    = "OMSGallery/ContainerInsights"
    }
}

resource "azurerm_kubernetes_cluster" "k8s" {
    name                = var.cluster_name
    location            = azurerm_resource_group.k8s.location
    resource_group_name = azurerm_resource_group.k8s.name
    dns_prefix          = var.dns_prefix

    linux_profile {
        admin_username = "ubuntu"

        ssh_key {
            key_data = file(var.ssh_public_key)
        }
    }

    default_node_pool {
        name            = "agentpool"
        node_count      = var.agent_count
        vm_size         = "Standard_DS1_v2"
    }

    service_principal {
        client_id     = var.client_id
        client_secret = var.client_secret
    }

    addon_profile {
        oms_agent {
            enabled            = true
            log_analytics_workspace_id = azurerm_log_analytics_workspace.test.id
        }
    }

    tags = {
        Environment = "Development"
    }
}

```

The preceding code sets the name of the cluster, location, and the resource group name. The prefix for the fully qualified domain name (FQDN) is also set. The FQDN is used to access the cluster.

The `linux_profile` record allows you to configure the settings that enable signing into the worker nodes using SSH.

With AKS, you pay only for the worker nodes. The `default_node_pool` record configures the details for these worker nodes. The `default_node_pool` record includes the number of worker nodes to create and the type of worker nodes. If you need to scale up or scale down the cluster in the future, you modify the `count` value in this record.

3. Save the file (<Ctrl>S) and exit the editor (<Ctrl>Q).

## Declare the variables

1. In Cloud Shell, create a file named `variables.tf`.

```
code variables.tf
```

2. Paste the following code into the editor:

```

variable "client_id" {}
variable "client_secret" {}

variable "agent_count" {
  default = 3
}

variable "ssh_public_key" {
  default = "~/.ssh/id_rsa.pub"
}

variable "dns_prefix" {
  default = "k8stest"
}

variable cluster_name {
  default = "k8stest"
}

variable resource_group_name {
  default = "azure-k8stest"
}

variable location {
  default = "Central US"
}

variable log_analytics_workspace_name {
  default = "testLogAnalyticsWorkspaceName"
}

# refer https://azure.microsoft.com/global-infrastructure/services/?products=monitor for log analytics
available regions
variable log_analytics_workspace_location {
  default = "eastus"
}

# refer https://azure.microsoft.com/pricing/details/monitor/ for log analytics pricing
variable log_analytics_workspace_sku {
  default = "PerGB2018"
}

```

3. Save the file (<Ctrl>S) and exit the editor (<Ctrl>Q).

## Create a Terraform output file

[Terraform outputs](#) allow you to define values that will be highlighted to the user when Terraform applies a plan, and can be queried using the `terraform output` command. In this section, you create an output file that allows access to the cluster with [kubectl](#).

1. In Cloud Shell, create a file named `output.tf`.

```
code output.tf
```

2. Paste the following code into the editor:

```

output "client_key" {
  value = azurerm_kubernetes_cluster.k8s.kube_config.0.client_key
}

output "client_certificate" {
  value = azurerm_kubernetes_cluster.k8s.kube_config.0.client_certificate
}

output "cluster_ca_certificate" {
  value = azurerm_kubernetes_cluster.k8s.kube_config.0.cluster_ca_certificate
}

output "cluster_username" {
  value = azurerm_kubernetes_cluster.k8s.kube_config.0.username
}

output "cluster_password" {
  value = azurerm_kubernetes_cluster.k8s.kube_config.0.password
}

output "kube_config" {
  value = azurerm_kubernetes_cluster.k8s.kube_config_raw
}

output "host" {
  value = azurerm_kubernetes_cluster.k8s.kube_config.0.host
}

```

3. Save the file (<Ctrl>S) and exit the editor (<Ctrl>Q).

## Set up Azure storage to store Terraform state

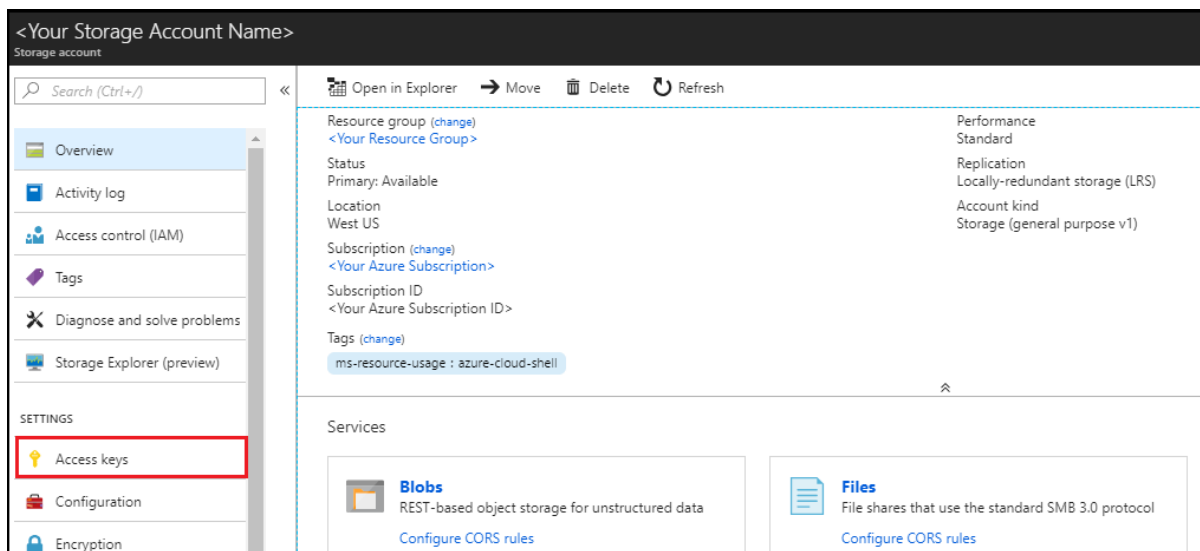
Terraform tracks state locally via the `terraform.tfstate` file. This pattern works well in a single-person environment. In a multi-person environment, [Azure storage](#) is used to track state.

In this section, you see how to do the following tasks:

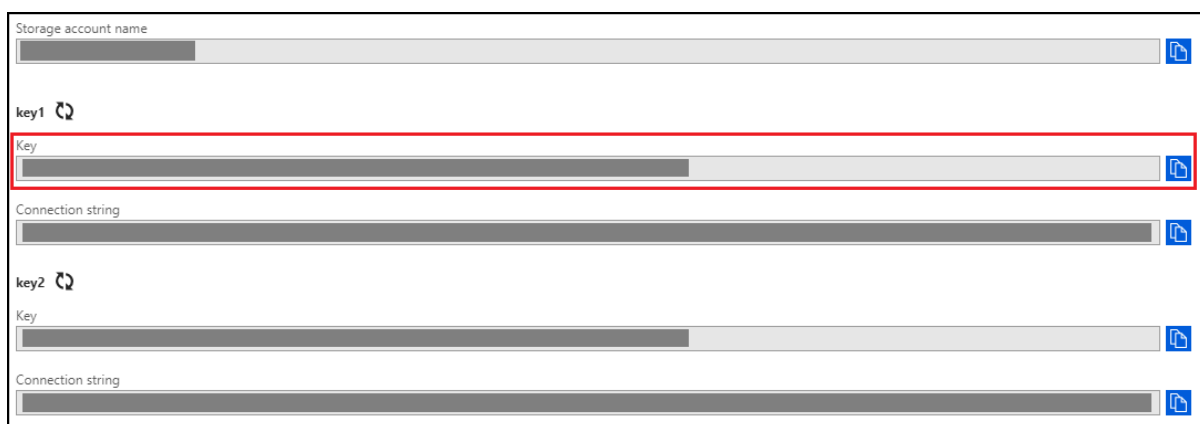
- Retrieve storage account information (account name and account key)
- Create a storage container into which Terraform state information will be stored.

1. In the Azure portal, select **All services** in the left menu.
2. Select **Storage accounts**.
3. On the **Storage accounts** tab, select the name of the storage account into which Terraform is to store state. For example, you can use the storage account created when you opened Cloud Shell the first time. The storage account name created by Cloud Shell typically starts with `cs` followed by a random string of numbers and letters. Take note of the storage account you select. This value is needed later.
4. On the storage account tab, select **Access keys**.





5. Make note of the **key1** key value. (Selecting the icon to the right of the key copies the value to the clipboard.)



6. In Cloud Shell, create a container in your Azure storage account. Replace the placeholders with appropriate values for your environment.

```
az storage container create -n tfstate --account-name <YourAzureStorageAccountName> --account-key <YourAzureStorageAccountKey>
```

## Create the Kubernetes cluster

In this section, you see how to use the `terraform init` command to create the resources defined in the configuration files you created in the previous sections.

1. In Cloud Shell, initialize Terraform. Replace the placeholders with appropriate values for your environment.

```
terraform init -backend-config="storage_account_name=<YourAzureStorageAccountName>" -backend-config="container_name=tfstate" -backend-config="access_key=<YourStorageAccountAccessKey>" -backend-config="key=key=code-lab.microsoft.tfstate"
```

The `terraform init` command displays the success of initializing the backend and provider plug-in:

```
Initializing the backend...

Initializing provider plugins...

Terraform has been successfully initialized!

You may now begin working with Terraform. Try running "terraform plan" to see
any changes that are required for your infrastructure. All Terraform commands
should now work.

If you ever set or change modules or backend configuration for Terraform,
rerun this command to reinitialize your working directory. If you forget, other
commands will detect it and remind you to do so if necessary.
```

2. Export your service principal credentials. Replace the placeholders with appropriate values from your service principal.

```
export TF_VAR_client_id=<service-principal-appid>
export TF_VAR_client_secret=<service-principal-password>
```

3. Run the `terraform plan` command to create the Terraform plan that defines the infrastructure elements.

```
terraform plan -out out.plan
```

The `terraform plan` command displays the resources that will be created when you run the `terraform apply` command:

```
id: <computed>
location: "centralus"
name: "azure-k8stest"
tags.%: <computed>

Plan: 2 to add, 0 to change, 0 to destroy.

-----

This plan was saved to: out.plan

To perform exactly these actions, run the following command to apply:
  terraform apply "out.plan"
```

4. Run the `terraform apply` command to apply the plan to create the Kubernetes cluster. The process to create a Kubernetes cluster can take several minutes, resulting in the Cloud Shell session timing out. If the Cloud Shell session times out, you can follow the steps in the section "Recover from a Cloud Shell timeout" to enable you to complete the tutorial.

```
terraform apply out.plan
```

The `terraform apply` command displays the results of creating the resources defined in your configuration files:

```

tags.%: "" => "<computed>"
azurerm_resource_group.k8s: Creation complete after 1s (ID: /subscriptions/ad7af7
azurerm_kubernetes_cluster.k8s: Creating...
agent_pool_profile.#: "" => "1"
agent_pool_profile.0.count: "" => "3"
agent_pool_profile.0.dns_prefix: "" => "<computed>"
agent_pool_profile.0.fqdn: "" => "<computed>"
agent_pool_profile.0.name: "" => "default"
agent_pool_profile.0.os_disk_size_gb: "" => "30"
agent_pool_profile.0.os_type: "" => "Linux"
agent_pool_profile.0.vm_size: "" => "Standard_D2"
dns_prefix: "" => "k8stest"
fqdn: "" => "<computed>"
kube_config.#: "" => "<computed>"
kube_config_raw: "<sensitive>" => "<sensitive>"
kubernetes_version: "" => "<computed>"
linux_profile.#: "" => "1"
linux_profile.0.admin_username: "" => "ubuntu"
linux_profile.0.ssh_key.#: "" => "1"
linux_profile.0.ssh_key.0.key_data: "" => "ssh-rsa"
location: "" => "centralus"
name: "" => "k8stest"
resource_group_name: "" => "azure-k8stest"
service_principal.#: "" => "1"
service_principal.2782116410.client_id: "" => ""
service_principal.2782116410.client_secret: "<sensitive>" => "<sensitive>"
tags.%: "" => "1"
tags.Environment: "" => "Development"
azurerm_kubernetes_cluster.k8s: Still creating... (10s elapsed)
azurerm_kubernetes_cluster.k8s: Still creating... (20s elapsed)
azurerm_kubernetes_cluster.k8s: Still creating... (30s elapsed)
azurerm_kubernetes_cluster.k8s: Still creating... (40s elapsed)
azurerm_kubernetes_cluster.k8s: Still creating... (50s elapsed)
azurerm_kubernetes_cluster.k8s: Still creating... (1m0s elapsed)

```

5. In the Azure portal, select **All resources** in the left menu to see the resources created for your new Kubernetes cluster.

Home > All resources

All resources

Microsoft

+ Add Edit columns Refresh Assign tags Delete

Subscriptions: 1 of 6 selected

Filter by name... Azure Subscription All resource groups All types All locations

48 items Show hidden types

NAME	RESOURCE GROUP	LOCATION
aks-agentpool-21324540-nsg	MC_nic-k8stest_k8stest_centralus	Central US
aks-agentpool-21324540-routetable	MC_nic-k8stest_k8stest_centralus	Central US
aks-default-21324540-0	MC_nic-k8stest_k8stest_centralus	Central US
aks-default-21324540-0_OsDisk_1_afcbe8a12cff4ed78ec32813cea6204a	MC_NIC-K8STEST_K8STEST_CENTRALUS	Central US
aks-default-21324540-1	MC_nic-k8stest_k8stest_centralus	Central US
aks-default-21324540-1_OsDisk_1_95c4a19caec8404f974868c84fa04523	MC_NIC-K8STEST_K8STEST_CENTRALUS	Central US
aks-default-21324540-2	MC_nic-k8stest_k8stest_centralus	Central US
aks-default-21324540-2_OsDisk_1_fb9917ab0b8f4ca0b901465819360f75	MC_NIC-K8STEST_K8STEST_CENTRALUS	Central US
aks-default-21324540-nic-0	MC_nic-k8stest_k8stest_centralus	Central US
aks-default-21324540-nic-1	MC_nic-k8stest_k8stest_centralus	Central US
aks-default-21324540-nic-2	MC_nic-k8stest_k8stest_centralus	Central US

## Recover from a Cloud Shell timeout

If the Cloud Shell session times out, you can do the following steps to recover:

1. Start a Cloud Shell session.
2. Change to the directory containing your Terraform configuration files.

```
cd /clouddrive/terraform-aks-k8s
```

3. Run the following command:

```
export KUBECONFIG=./azurek8s
```

## Test the Kubernetes cluster

The Kubernetes tools can be used to verify the newly created cluster.

1. Get the Kubernetes configuration from the Terraform state and store it in a file that kubectl can read.

```
echo "$(terraform output kube_config)" > ./azurek8s
```

2. Set an environment variable so that kubectl picks up the correct config.

```
export KUBECONFIG=./azurek8s
```

3. Verify the health of the cluster.

```
kubectl get nodes
```

You should see the details of your worker nodes, and they should all have a status **Ready**, as shown in the following image:

```
$ kubectl get nodes
```

NAME	STATUS	ROLES	AGE	VERSION
aks-default-27881813-0	Ready	agent	48m	v1.9.6
aks-default-27881813-1	Ready	agent	48m	v1.9.6
aks-default-27881813-2	Ready	agent	48m	v1.9.6

## Monitor health and logs

When the AKS cluster was created, monitoring was enabled to capture health metrics for both the cluster nodes and pods. These health metrics are available in the Azure portal. For more information on container health monitoring, see [Monitor Azure Kubernetes Service health](#).

## Next steps

[Learn more about using Terraform in Azure](#)

[Azure Kubernetes Service \(AKS\)](#) manages your hosted Kubernetes environment. AKS makes it quick and easy to deploy and manage containerized applications without container orchestration expertise. AKS also eliminates the burden of taking applications offline for operational and maintenance tasks. Using AKS, these tasks - including provisioning, upgrading and scaling resources - can be accomplished on-demand.

An ingress controller provides various features for Kubernetes services. These features include reverse proxy, configurable traffic routing, and TLS termination. Kubernetes ingress resources are used to configure the ingress rules for individual Kubernetes services. Using an ingress controller and ingress rules, a single IP address can route traffic to multiple services in a Kubernetes cluster. All this functionality is provided by Azure [Application Gateway](#), making it an ideal Ingress controller for Kubernetes on Azure.

In this tutorial, you learn how to do the following tasks:

- Create a [Kubernetes](#) cluster using AKS with Application Gateway as Ingress Controller.
- Use HCL (HashiCorp Language) to define a Kubernetes cluster.
- Use Terraform to create Application Gateway resource.
- Use Terraform and AKS to create a Kubernetes cluster.
- Use the kubectl tool to test the availability of a Kubernetes cluster.

## Prerequisites

- **Azure subscription:** If you don't have an Azure subscription, create a [free account](#) before you begin.
- **Configure Terraform:** Follow the directions in the article, [Terraform and configure access to Azure](#)
- **Azure resource group:** If you don't have an Azure resource group to use for the demo, [create an Azure resource group](#). Take note of the resource group name and location as those values are used in the demo.
- **Azure service principal:** Follow the directions in the section of the **Create the service principal** section in the article, [Create an Azure service principal with Azure CLI](#). Take note of the values for the appId, displayName, and password.
- **Obtain the Service Principal Object ID:** Run the following command in Cloud Shell:

```
az ad sp list --display-name <displayName>
```

## Create the directory structure

The first step is to create the directory that holds your Terraform configuration files for the exercise.

1. Browse to the [Azure portal](#).
2. Open [Azure Cloud Shell](#).
3. Change directories to the `clouddrive` directory.

```
cd clouddrive
```

4. Create a directory named `terraform-aks-appgw-ingress`.

```
mkdir terraform-aks-appgw-ingress
```

5. Change directories to the new directory:

```
cd terraform-aks-appgw-ingress
```

## Declare the Azure provider

Create the Terraform configuration file that declares the Azure provider.

1. In Cloud Shell, create a file named `main.tf`.

```
code main.tf
```

2. Paste the following code into the editor:

```
provider "azurerm" {  
  # The "feature" block is required for AzureRM provider 2.x.  
  # If you are using version 1.x, the "features" block is not allowed.  
  version = "~>2.0"  
  features {}  
}  
  
terraform {  
  backend "azurerm" {}  
}
```

3. Save the file (<Ctrl>S) and exit the editor (<Ctrl>Q).

## Define input variables

Create the Terraform configuration file that lists all the variables required for this deployment.

1. In Cloud Shell, create a file named `variables.tf`.

```
code variables.tf
```

2. Paste the following code into the editor:

```
variable "resource_group_name" {  
  description = "Name of the resource group."  
}  
  
variable "location" {  
  description = "Location of the cluster."  
}  
  
variable "aks_service_principal_app_id" {  
  description = "Application ID/Client ID of the service principal. Used by AKS to manage AKS related  
resources on Azure like vms, subnets."  
}  
  
variable "aks_service_principal_client_secret" {  
  description = "Secret of the service principal. Used by AKS to manage Azure."  
}  
  
variable "aks_service_principal_object_id" {
```

```

variable "aks_service_principal_object_id" {
  description = "Object ID of the service principal."
}

variable "virtual_network_name" {
  description = "Virtual network name"
  default     = "aksVirtualNetwork"
}

variable "virtual_network_address_prefix" {
  description = "Containers DNS server IP address."
  default     = "15.0.0.0/8"
}

variable "aks_subnet_name" {
  description = "AKS Subnet Name."
  default     = "kubesubnet"
}

variable "aks_subnet_address_prefix" {
  description = "Containers DNS server IP address."
  default     = "15.0.0.0/16"
}

variable "app_gateway_subnet_address_prefix" {
  description = "Containers DNS server IP address."
  default     = "15.1.0.0/16"
}

variable "app_gateway_name" {
  description = "Name of the Application Gateway."
  default     = "ApplicationGateway1"
}

variable "app_gateway_sku" {
  description = "Name of the Application Gateway SKU."
  default     = "Standard_v2"
}

variable "app_gateway_tier" {
  description = "Tier of the Application Gateway SKU."
  default     = "Standard_v2"
}

variable "aks_name" {
  description = "Name of the AKS cluster."
  default     = "aks-cluster1"
}

variable "aks_dns_prefix" {
  description = "Optional DNS prefix to use with hosted Kubernetes API server FQDN."
  default     = "aks"
}

variable "aks_agent_os_disk_size" {
  description = "Disk size (in GB) to provision for each of the agent pool nodes. This value ranges from 0 to 1023. Specifying 0 applies the default disk size for that agentVMSize."
  default     = 40
}

variable "aks_agent_count" {
  description = "The number of agent nodes for the cluster."
  default     = 3
}

variable "aks_agent_vm_size" {
  description = "The size of the Virtual Machine."
  default     = "Standard_D3_v2"
}

```

```

variable "kubernetes_version" {
  description = "The version of Kubernetes."
  default     = "1.11.5"
}

variable "aks_service_cidr" {
  description = "A CIDR notation IP range from which to assign service cluster IPs."
  default     = "10.0.0.0/16"
}

variable "aks_dns_service_ip" {
  description = "Containers DNS server IP address."
  default     = "10.0.0.10"
}

variable "aks_docker_bridge_cidr" {
  description = "A CIDR notation IP for Docker bridge."
  default     = "172.17.0.1/16"
}

variable "aks_enable_rbac" {
  description = "Enable RBAC on the AKS cluster. Defaults to false."
  default     = "false"
}

variable "vm_user_name" {
  description = "User name for the VM"
  default     = "vmuser1"
}

variable "public_ssh_key_path" {
  description = "Public key path for SSH."
  default     = "~/ssh/id_rsa.pub"
}

variable "tags" {
  type = "map"

  default = {
    source = "terraform"
  }
}

```

3. Save the file (<Ctrl>S) and exit the editor (<Ctrl>Q).

## Define the resources

Create Terraform configuration file that creates all the resources.

1. In Cloud Shell, create a file named `resources.tf`.

```
code resources.tf
```

2. Paste the following code block to create a locals block for computed variables to reuse:



```
# # Locals block for hardcoded names.
locals {
  backend_address_pool_name      = "${azurerm_virtual_network.test.name}-beap"
  frontend_port_name             = "${azurerm_virtual_network.test.name}-feport"
  frontend_ip_configuration_name = "${azurerm_virtual_network.test.name}-feip"
  http_setting_name              = "${azurerm_virtual_network.test.name}-be-htst"
  listener_name                  = "${azurerm_virtual_network.test.name}-httplstn"
  request_routing_rule_name      = "${azurerm_virtual_network.test.name}-rqrt"
  app_gateway_subnet_name       = "appgwsubnet"
}
```

3. Paste the following code block to create a data source for Resource group, new User identity:

```
data "azurerm_resource_group" "rg" {
  name = var.resource_group_name
}

# User Assigned Identities
resource "azurerm_user_assigned_identity" "testIdentity" {
  resource_group_name = data.azurerm_resource_group.rg.name
  location             = data.azurerm_resource_group.rg.location

  name = "identity1"

  tags = var.tags
}
```

4. Paste the following code block to create base networking resources:

```

resource "azurerm_virtual_network" "test" {
  name                = var.virtual_network_name
  location            = data.azurerm_resource_group.rg.location
  resource_group_name = data.azurerm_resource_group.rg.name
  address_space       = [var.virtual_network_address_prefix]

  subnet {
    name                = var.aks_subnet_name
    address_prefix      = var.aks_subnet_address_prefix
  }

  subnet {
    name                = "appgws subnet"
    address_prefix      = var.app_gateway_subnet_address_prefix
  }

  tags = var.tags
}

data "azurerm_subnet" "kubesubnet" {
  name                = var.aks_subnet_name
  virtual_network_name = azurerm_virtual_network.test.name
  resource_group_name = data.azurerm_resource_group.rg.name
}

data "azurerm_subnet" "appgws subnet" {
  name                = "appgws subnet"
  virtual_network_name = azurerm_virtual_network.test.name
  resource_group_name = data.azurerm_resource_group.rg.name
}

# Public Ip
resource "azurerm_public_ip" "test" {
  name                = "publicIp1"
  location            = data.azurerm_resource_group.rg.location
  resource_group_name = data.azurerm_resource_group.rg.name
  allocation_method   = "Static"
  sku                 = "Standard"

  tags = var.tags
}

```

5. Paste the following code block to create Application Gateway resource:

```

resource "azurerm_application_gateway" "network" {
  name                = var.app_gateway_name
  resource_group_name = data.azurerm_resource_group.rg.name
  location            = data.azurerm_resource_group.rg.location

  sku {
    name      = var.app_gateway_sku
    tier       = "Standard_v2"
    capacity  = 2
  }

  gateway_ip_configuration {
    name      = "appGatewayIpConfig"
    subnet_id = data.azurerm_subnet.appgws subnet.id
  }

  frontend_port {
    name = local.frontend_port_name
    port = 80
  }

  frontend_port {
    name = "httpsPort"
    port = 443
  }

  frontend_ip_configuration {
    name                = local.frontend_ip_configuration_name
    public_ip_address_id = azurerm_public_ip.test.id
  }

  backend_address_pool {
    name = local.backend_address_pool_name
  }

  backend_http_settings {
    name                = local.http_setting_name
    cookie_based_affinity = "Disabled"
    port                = 80
    protocol             = "Http"
    request_timeout      = 1
  }

  http_listener {
    name                = local.listener_name
    frontend_ip_configuration_name = local.frontend_ip_configuration_name
    frontend_port_name      = local.frontend_port_name
    protocol                 = "Http"
  }

  request_routing_rule {
    name                = local.request_routing_rule_name
    rule_type           = "Basic"
    http_listener_name   = local.listener_name
    backend_address_pool_name = local.backend_address_pool_name
    backend_http_settings_name = local.http_setting_name
  }

  tags = var.tags

  depends_on = ["azurerm_virtual_network.test", "azurerm_public_ip.test"]
}

```

6. Paste the following code block to create role assignments:

```

resource "azurerm_role_assignment" "ra1" {
  scope                = data.azurerm_subnet.kubesubnet.id
  role_definition_name = "Network Contributor"
  principal_id         = var.aks_service_principal_object_id

  depends_on = ["azurerm_virtual_network.test"]
}

resource "azurerm_role_assignment" "ra2" {
  scope                = azurerm_user_assigned_identity.testIdentity.id
  role_definition_name = "Managed Identity Operator"
  principal_id         = var.aks_service_principal_object_id
  depends_on           = ["azurerm_user_assigned_identity.testIdentity"]
}

resource "azurerm_role_assignment" "ra3" {
  scope                = azurerm_application_gateway.network.id
  role_definition_name = "Contributor"
  principal_id         = azurerm_user_assigned_identity.testIdentity.principal_id
  depends_on           = ["azurerm_user_assigned_identity.testIdentity",
"azurerm_application_gateway.network"]
}

resource "azurerm_role_assignment" "ra4" {
  scope                = data.azurerm_resource_group.rg.id
  role_definition_name = "Reader"
  principal_id         = azurerm_user_assigned_identity.testIdentity.principal_id
  depends_on           = ["azurerm_user_assigned_identity.testIdentity",
"azurerm_application_gateway.network"]
}

```

7. Paste the following code block to create the Kubernetes cluster:

```

resource "azurerm_kubernetes_cluster" "k8s" {
  name          = var.aks_name
  location      = data.azurerm_resource_group.rg.location
  dns_prefix    = var.aks_dns_prefix

  resource_group_name = data.azurerm_resource_group.rg.name

  linux_profile {
    admin_username = var.vm_user_name

    ssh_key {
      key_data = file(var.public_ssh_key_path)
    }
  }

  addon_profile {
    http_application_routing {
      enabled = false
    }
  }

  default_node_pool {
    name          = "agentpool"
    node_count    = var.aks_agent_count
    vm_size       = var.aks_agent_vm_size
    os_disk_size_gb = var.aks_agent_os_disk_size
    vnet_subnet_id = data.azurerm_subnet.kubesubnet.id
  }

  service_principal {
    client_id     = var.aks_service_principal_app_id
    client_secret = var.aks_service_principal_client_secret
  }

  network_profile {
    network_plugin = "azure"
    dns_service_ip = var.aks_dns_service_ip
    docker_bridge_cidr = var.aks_docker_bridge_cidr
    service_cidr      = var.aks_service_cidr
  }

  depends_on = ["azurerm_virtual_network.test", "azurerm_application_gateway.network"]
  tags       = var.tags
}

```

8. Save the file (<Ctrl>S) and exit the editor (<Ctrl>Q).

The code presented in this section sets the name of the cluster, location, and the `resource_group_name`. The `dns_prefix` value - that forms part of the fully qualified domain name (FQDN) used to access the cluster - is set.

The `linux_profile` record allows you to configure the settings that enable signing into the worker nodes using SSH.

With AKS, you pay only for the worker nodes. The `agent_pool_profile` record configures the details for these worker nodes. The `agent_pool_profile` record includes the number of worker nodes to create and the type of worker nodes. If you need to scale up or scale down the cluster in the future, you modify the `count` value in this record.

## Create a Terraform output file

[Terraform outputs](#) allow you to define values that are highlighted to the user when Terraform applies a plan, and can be queried using the `terraform output` command. In this section, you create an output file that allows access to

the cluster with [kubectl](#).

1. In Cloud Shell, create a file named `output.tf`.

```
code output.tf
```

2. Paste the following code into the editor:

```
output "client_key" {
  value = azurerm_kubernetes_cluster.k8s.kube_config.0.client_key
}

output "client_certificate" {
  value = azurerm_kubernetes_cluster.k8s.kube_config.0.client_certificate
}

output "cluster_ca_certificate" {
  value = azurerm_kubernetes_cluster.k8s.kube_config.0.cluster_ca_certificate
}

output "cluster_username" {
  value = azurerm_kubernetes_cluster.k8s.kube_config.0.username
}

output "cluster_password" {
  value = azurerm_kubernetes_cluster.k8s.kube_config.0.password
}

output "kube_config" {
  value = azurerm_kubernetes_cluster.k8s.kube_config_raw
}

output "host" {
  value = azurerm_kubernetes_cluster.k8s.kube_config.0.host
}

output "identity_resource_id" {
  value = azurerm_user_assigned_identity.testIdentity.id
}

output "identity_client_id" {
  value = azurerm_user_assigned_identity.testIdentity.client_id
}
```

3. Save the file (<Ctrl>S) and exit the editor (<Ctrl>Q).

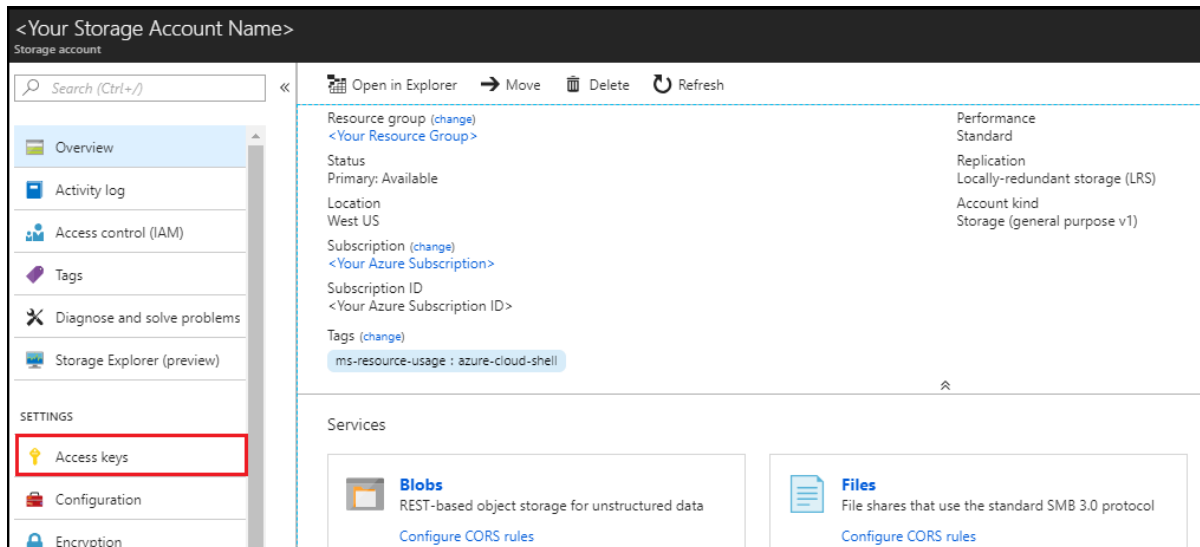
## Configure Azure storage to store Terraform state

Terraform tracks state locally via the `terraform.tfstate` file. This pattern works well in a single-person environment. However, in a more practical multi-person environment, you need to track state on the server using [Azure storage](#). In this section, you learn to retrieve the necessary storage account information and create a storage container. The Terraform state information is then stored in that container.

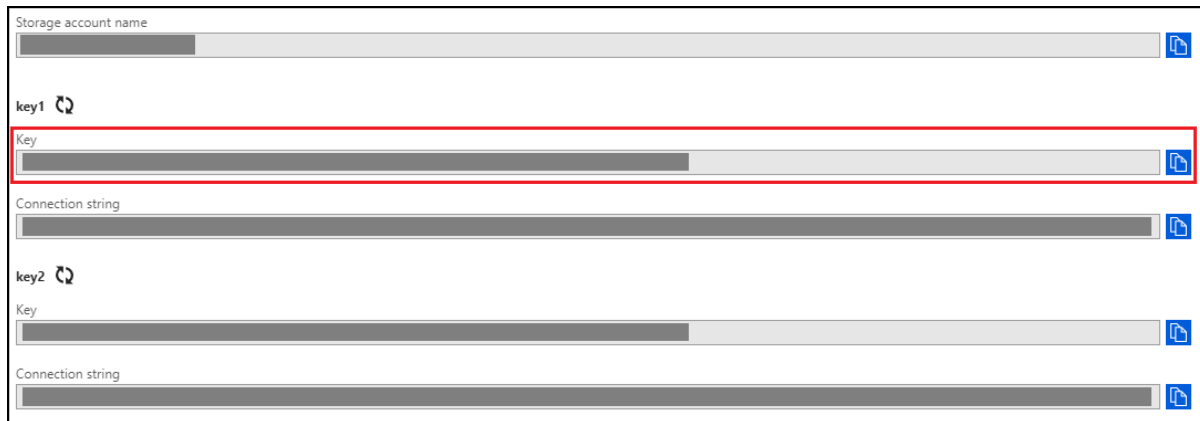
1. In the Azure portal, under **Azure services**, select **Storage accounts**. (If the **Storage accounts** option isn't visible on the main page, select **More services** and then locate and select it.)
2. On the **Storage accounts** page, select the name of the storage account into which Terraform is to store state. For example, you can use the storage account created when you opened Cloud Shell the first time. The storage account name created by Cloud Shell typically starts with `cs` followed by a random string of numbers and letters.

Take note of the storage account you select, as you need it later.

3. On the storage account page, select **Access keys**.



4. Make note of the **key1** key value. (Selecting the icon to the right of the key copies the value to the clipboard.)



5. In Cloud Shell, create a container in your Azure storage account. Replace the placeholders with the appropriate values for your Azure storage account.

```
az storage container create -n tfstate --account-name <YourAzureStorageAccountName> --account-key <YourAzureStorageAccountKey>
```

## Create the Kubernetes cluster

In this section, you see how to use the `terraform init` command to create the resources defined in the configuration files you created in the previous sections.

1. In Cloud Shell, initialize Terraform. Replace the placeholders with the appropriate values for your Azure storage account.

```
terraform init -backend-config="storage_account_name=<YourAzureStorageAccountName>" -backend-config="container_name=tfstate" -backend-config="access_key=<YourStorageAccountAccessKey>" -backend-config="key=codelab.microsoft.tfstate"
```

The `terraform init` command displays the success of initializing the backend and provider plug-in:

```
Initializing the backend...
```

```
Initializing provider plugins...
```

```
Terraform has been successfully initialized!
```

```
You may now begin working with Terraform. Try running "terraform plan" to see any changes that are required for your infrastructure. All Terraform commands should now work.
```

```
If you ever set or change modules or backend configuration for Terraform, rerun this command to reinitialize your working directory. If you forget, other commands will detect it and remind you to do so if necessary.
```

2. In Cloud Shell, create a file named `terraform.tfvars`:

```
code terraform.tfvars
```

3. Paste the following variables created earlier into the editor. To get the location value for your environment, use `az account list-locations`.

```
resource_group_name = "<Name of the Resource Group already created>"

location = "<Location of the Resource Group>"

aks_service_principal_app_id = "<Service Principal AppId>"

aks_service_principal_client_secret = "<Service Principal Client Secret>"

aks_service_principal_object_id = "<Service Principal Object Id>"
```

4. Save the file (<Ctrl>S) and exit the editor (<Ctrl>Q).
5. Run the `terraform plan` command to create the Terraform plan that defines the infrastructure elements.

```
terraform plan -out out.plan
```

The `terraform plan` command displays the resources that are created when you run the `terraform apply` command:

```
id:                <computed>
location:          "centralus"
name:              "azure-k8stest"
tags.%:            <computed>
```

```
Plan: 2 to add, 0 to change, 0 to destroy.
```

```
-----
This plan was saved to: out.plan
```

```
To perform exactly these actions, run the following command to apply:
  terraform apply "out.plan"
```



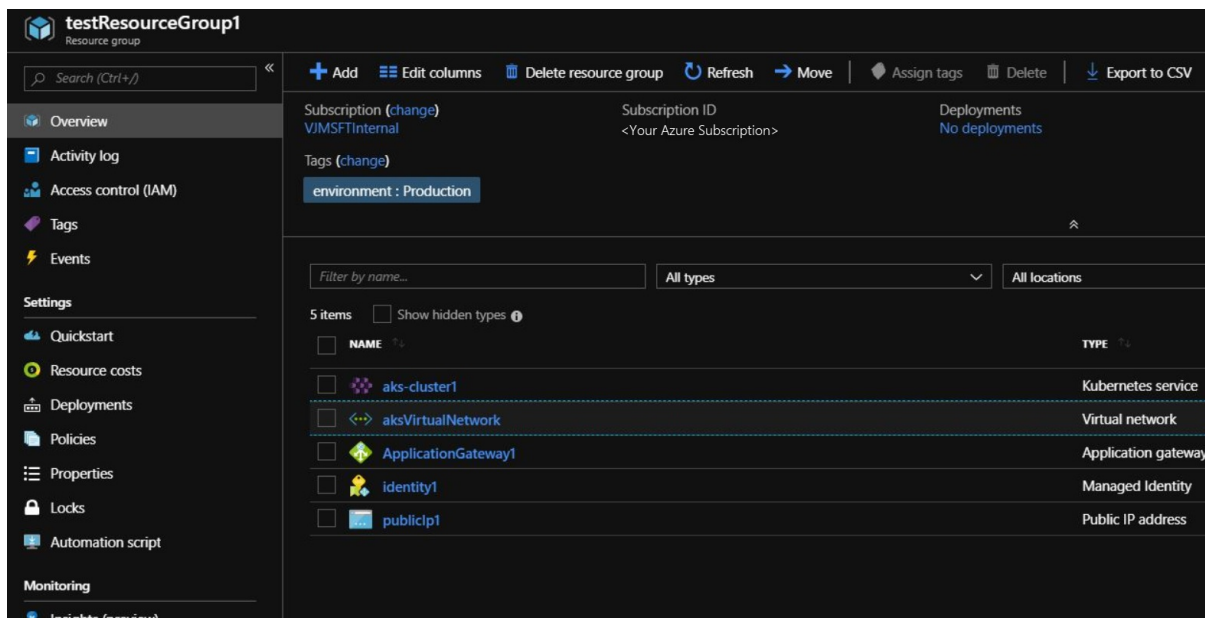
6. Run the `terraform apply` command to apply the plan to create the Kubernetes cluster. The process to create a Kubernetes cluster can take several minutes, resulting in the Cloud Shell session timing out. If the Cloud Shell session times out, you can follow the steps in the section "Recover from a Cloud Shell timeout" to enable you to complete the tutorial.

```
terraform apply out.plan
```

The `terraform apply` command displays the results of creating the resources defined in your configuration files:

```
tags.%: "" => "<computed>"
azurerm_resource_group.k8s: Creation complete after 1s (ID: /subscriptions/ad7af7
azurerm_kubernetes_cluster.k8s: Creating...
  agent_pool_profile.#: "" => "1"
  agent_pool_profile.0.count: "" => "3"
  agent_pool_profile.0.dns_prefix: "" => "<computed>"
  agent_pool_profile.0.fqdn: "" => "<computed>"
  agent_pool_profile.0.name: "" => "default"
  agent_pool_profile.0.os_disk_size_gb: "" => "30"
  agent_pool_profile.0.os_type: "" => "Linux"
  agent_pool_profile.0.vm_size: "" => "Standard_D2"
  dns_prefix: "" => "k8stest"
  fqdn: "" => "<computed>"
  kube_config.#: "" => "<computed>"
  kube_config_raw: "<sensitive>" => "<sensitive>"
  kubernetes_version: "" => "<computed>"
  linux_profile.#: "" => "1"
  linux_profile.0.admin_username: "" => "ubuntu"
  linux_profile.0.ssh_key.#: "" => "1"
  linux_profile.0.ssh_key.0.key_data: "" => "ssh-rsa"
  location: "" => "centralus"
  name: "" => "k8stest"
  resource_group_name: "" => "azure-k8stest"
  service_principal.#: "" => "1"
  service_principal.2782116410.client_id: "" => ""
  service_principal.2782116410.client_secret: "<sensitive>" => "<sensitive>"
  tags.%: "" => "1"
  tags.Environment: "" => "Development"
azurerm_kubernetes_cluster.k8s: Still creating... (10s elapsed)
azurerm_kubernetes_cluster.k8s: Still creating... (20s elapsed)
azurerm_kubernetes_cluster.k8s: Still creating... (30s elapsed)
azurerm_kubernetes_cluster.k8s: Still creating... (40s elapsed)
azurerm_kubernetes_cluster.k8s: Still creating... (50s elapsed)
azurerm_kubernetes_cluster.k8s: Still creating... (1m0s elapsed)
```

7. In the Azure portal, select **Resource Groups** in the left menu to see the resources created for your new Kubernetes cluster in the selected resource group.



## Recover from a Cloud Shell timeout

If the Cloud Shell session times out, you can use the following steps to recover:

1. Start a Cloud Shell session.
2. Change to the directory containing your Terraform configuration files.

```
cd /clouddrive/terraform-aks-k8s
```

3. Run the following command:

```
export KUBECONFIG=./azurek8s
```

## Test the Kubernetes cluster

The Kubernetes tools can be used to verify the newly created cluster.

1. Get the Kubernetes configuration from the Terraform state and store it in a file that kubectl can read.

```
echo "${terraform output kube_config}" > ./azurek8s
```

2. Set an environment variable so that kubectl picks up the correct config.

```
export KUBECONFIG=./azurek8s
```

3. Verify the health of the cluster.

```
kubectl get nodes
```

You should see the details of your worker nodes, and they should all have a status **Ready**, as shown in the following image:

```
$ kubectl get nodes
```

NAME	STATUS	ROLES	AGE	VERSION
aks-default-27881813-0	Ready	agent	48m	v1.9.6
aks-default-27881813-1	Ready	agent	48m	v1.9.6
aks-default-27881813-2	Ready	agent	48m	v1.9.6

## Install Azure AD Pod Identity

Azure Active Directory Pod Identity provides token-based access to [Azure Resource Manager](#).

[Azure AD Pod Identity](#) adds the following components to your Kubernetes cluster:

- Kubernetes **CRDs**: `AzureIdentity`, `AzureAssignedIdentity`, `AzureIdentityBinding`
- [Managed Identity Controller \(MIC\)](#) component
- [Node Managed Identity \(NMI\)](#) component

If RBAC is **enabled**, run the following command to install Azure AD Pod Identity to your cluster:

```
kubectl create -f https://raw.githubusercontent.com/Azure/aad-pod-identity/master/deploy/infra/deployment-rbac.yaml
```

If RBAC is **disabled**, run the following command to install Azure AD Pod Identity to your cluster:

```
kubectl create -f https://raw.githubusercontent.com/Azure/aad-pod-identity/master/deploy/infra/deployment.yaml
```

## Install Helm

The code in this section uses [Helm](#) - Kubernetes package manager - to install the

`application-gateway-kubernetes-ingress` package:

1. If RBAC is **enabled**, run the following set of commands to install and configure Helm:

```
kubectl create serviceaccount --namespace kube-system tiller-sa
kubectl create clusterrolebinding tiller-cluster-rule --clusterrole=cluster-admin --serviceaccount=kube-system:tiller-sa
helm init --tiller-namespace kube-system --service-account tiller-sa
```

2. If RBAC is **disabled**, run the following command to install and configure Helm:

```
helm init
```

3. Add the AGIC Helm repository:

```
helm repo add application-gateway-kubernetes-ingress https://appgwingress.blob.core.windows.net/ingress-azure-helm-package/
helm repo update
```

## Install Ingress Controller Helm Chart

1. Download `helm-config.yaml` to configure AGIC:

```
wget https://raw.githubusercontent.com/Azure/application-gateway-kubernetes-ingress/master/docs/examples/sample-helm-config.yaml -O helm-config.yaml
```

2. Edit the `helm-config.yaml` and enter appropriate values for `appgw` and `armAuth` sections.

```
code helm-config.yaml
```

The values are described as follows:

- `verbosityLevel`: Sets the verbosity level of the AGIC logging infrastructure. See [Logging Levels](#) for possible values.
- `appgw.subscriptionId`: The Azure Subscription ID for the App Gateway. Example: `a123b234-a3b4-557d-b2df-a0bc12de1234`
- `appgw.resourceGroup`: Name of the Azure Resource Group in which App Gateway was created.
- `appgw.name`: Name of the Application Gateway. Example: `applicationgateway1`.
- `appgw.shared`: This boolean flag should be defaulted to `false`. Set to `true` should you need a [Shared App Gateway](#).
- `kubernetes.watchNamespace`: Specify the name space, which AGIC should watch. The namespace can be a single string value, or a comma-separated list of namespaces. Leaving this variable commented out, or setting it to blank or empty string results in Ingress Controller observing all accessible namespaces.
- `armAuth.type`: A value of either `aadPodIdentity` or `servicePrincipal`.
- `armAuth.identityResourceID`: Resource ID of the managed identity.
- `armAuth.identityClientId`: The Client ID of the Identity.
- `armAuth.secretJSON`: Only needed when Service Principal Secret type is chosen (when `armAuth.type` has been set to `servicePrincipal`).

Key notes:

- The `identityResourceID` value is created in the terraform script and can be found by running:  

```
echo "$(terraform output identity_resource_id)"
```
- The `identityClientId` value is created in the terraform script and can be found by running:  

```
echo "$(terraform output identity_client_id)"
```
- The `<resource-group>` value is the resource group of your App Gateway.
- The `<identity-name>` value is the name of the created identity.
- All identities for a given subscription can be listed using: `az identity list`.

3. Install the Application Gateway ingress controller package:

```
helm install -f helm-config.yaml application-gateway-kubernetes-ingress/ingress-azure
```

## Install a sample app

Once you have the App Gateway, AKS, and AGIC installed, you can install a sample app via [Azure Cloud Shell](#):

1. Use the curl command to download the YAML file:

```
curl https://raw.githubusercontent.com/Azure/application-gateway-kubernetes-ingress/master/docs/examples/aspnetapp.yaml -o aspnetapp.yaml
```

2. Apply the YAML file:

```
kubectl apply -f aspnetapp.yaml
```

## Clean up resources

When no longer needed, delete the resources created in this article.

Replace the placeholder with the appropriate value. All resources within the specified resource group will be deleted.

```
az group delete -n <resource-group>
```

## Next steps

[Application Gateway Ingress Controller](#)

This article walks you through creating a small VM cluster with the Terraform [Azure compute module](#). In this tutorial you learn how to:

- Set up authentication with Azure
- Create the Terraform template
- Visualize the changes with plan
- Apply the configuration to create the VM cluster

For more information on Terraform, see the [Terraform documentation](#).

## Set up authentication with Azure

### TIP

If you [use Terraform environment variables](#) or run this tutorial in the [Azure Cloud Shell](#), skip this step.

Review [Install Terraform and configure access to Azure](#) to create an Azure service principal. Use this service principal to populate a new file `azureProviderAndCreds.tf` in an empty directory with the following code:

```
variable subscription_id {}
variable tenant_id {}
variable client_id {}
variable client_secret {}

provider "azurerm" {
  version = "~>1.40"

  subscription_id = "xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxxxx"
  tenant_id       = "xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxxxx"
  client_id       = "xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxxxx"
  client_secret   = "xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxxxx"
}
```

## Create the template

Create a new Terraform template named `main.tf` with the following code:

```

module mycompute {
  source = "Azure/compute/azurerm"
  resource_group_name = "myResourceGroup"
  location = "East US 2"
  admin_password = "ComplxP@assw0rd!"
  vm_os_simple = "WindowsServer"
  is_windows_image = "true"
  remote_port = "3389"
  nb_instances = 2
  public_ip_dns = ["unique_dns_name"]
  vnet_subnet_id = module.network.vnet_subnets[0]
}

module "network" {
  source = "Azure/network/azurerm"
  location = "East US 2"
  resource_group_name = "myResourceGroup"
}

output "vm_public_name" {
  value = module.mycompute.public_ip_dns_name
}

output "vm_public_ip" {
  value = module.mycompute.public_ip_address
}

output "vm_private_ips" {
  value = module.mycompute.network_interface_private_ip
}

```

Run `terraform init` in your configuration directory. Using a Terraform version of at least 0.10.6 shows the following output:

```

greg@terraformDevBox:~/tfconfig$
greg@terraformDevBox:~/tfconfig$
greg@terraformDevBox:~/tfconfig$ terraform init
Downloading modules...
Get: https://api.github.com/repos/Azure/terraform-azurerm-compute/tarball/v1.0.2?archive=tar.gz
Get: https://api.github.com/repos/Azure/terraform-azurerm-network/tarball/v1.0.1?archive=tar.gz
Get: file:///home/greg/tfconfig/.terraform/modules/edb8d6b31571fe5b1e2af73395963beb/Azure-terraform-azurerm-compute-4a4acdd/os
Initializing provider plugins...
Terraform has been successfully initialized!

You may now begin working with Terraform. Try running "terraform plan" to see
any changes that are required for your infrastructure. All Terraform commands
should now work.

If you ever set or change modules or backend configuration for Terraform,
rerun this command to reinitialize your working directory. If you forget, other
commands will detect it and remind you to do so if necessary.
greg@terraformDevBox:~/tfconfig$ █

```

## Visualize the changes with plan

Run `terraform plan` to preview the virtual machine infrastructure created by the template.

```

C:\terraform-test>terraform plan
Refreshing Terraform state in-memory prior to plan...
The refreshed state will be used to calculate this plan, but will not be
persisted to local or remote state storage.

-----

An execution plan has been generated and is shown below.
Resource actions are indicated with the following symbols:
+ create

Terraform will perform the following actions:

+ azurerm_availability_set.avset
  id:                               <computed>
  location:                         "westus2"
  managed:                          "true"
  name:                             "avset"
  platform_fault_domain_count:      "2"
  platform_update_domain_count:     "2"
  resource_group_name:              "acctestrg"
  tags.%:                           <computed>

+ azurerm_lb.test
  id:                               <computed>

```

## Create the virtual machines with apply

Run `terraform apply` to provision the VMs on Azure.

```

C:\terraform-test>terraform apply
azurerm_resource_group.test: Creating...
  location: "" => "westus2"
  name:     "" => "acctestrg"
  tags.%:   "" => "<computed>"
azurerm_resource_group.test: Creation complete after 0s (ID: /subscriptions/ad7af
azurerm_managed_disk.test[1]: Creating...
  create_option: "" => "Empty"
  disk_size_gb:  "" => "1023"
  location:      "" => "westus2"
  name:          "" => "datadisk_existing_1"
  resource_group_name: "" => "acctestrg"
  source_uri:     "" => "<computed>"
  storage_account_type: "" => "Standard_LRS"
  tags.%:        "" => "<computed>"
azurerm_virtual_network.test: Creating...
  address_space.#: "" => "1"
  address_space.0: "" => "10.0.0.0/16"
  location:        "" => "westus2"
  name:            "" => "acctvn"
  resource_group_name: "" => "acctestrg"
  subnet.#:        "" => "<computed>"
  tags.%:          "" => "<computed>"
azurerm_availability_set.avset: Creating...
  location: "" => "westus2"
  managed:  "" => "true"

```

## Next steps

[Browse the list of Azure Terraform modules](#)



In this tutorial, you see how to create a small compute cluster using [HCL](#).

You'll learn how to do the following tasks:

- Set up Azure authentication.
- Create a Terraform configuration file.
- Use a Terraform configuration file to create a load balancer.
- Use a Terraform configuration file to deploy two Linux VMs in an availability set.
- Initialize Terraform.
- Create a Terraform execution plan.
- Apply the Terraform execution plan to create the Azure resources.

## 1. Set up Azure authentication

### NOTE

If you [use Terraform environment variables](#), or run this tutorial in the [Azure Cloud Shell](#), skip this section.

In this section, you generate an Azure service principal, and two Terraform configuration files containing the credentials from the security principal.

1. [Set up an Azure AD service principal](#) to enable Terraform to provision resources into Azure. While creating the principal, Make note of the values for the subscription ID, tenant, appId, and password.
2. Open a command prompt.
3. Create an empty directory in which to store your Terraform files.
4. Create a new file that holds your variables declarations. You can name this file anything you like with a `.tf` extension.
5. Copy the following code into your variable declaration file:

```
variable subscription_id {}
variable tenant_id {}
variable client_id {}
variable client_secret {}

provider "azurerm" {
  version = "~>1.40"

  subscription_id = var.subscription_id
  tenant_id = var.tenant_id
  client_id = var.client_id
  client_secret = var.client_secret
}
```

6. Create a new file that contains the values for your Terraform variables. It's common to name your Terraform variable file `terraform.tfvars` as Terraform automatically loads any file named `terraform.tfvars` (or following a pattern of `*.auto.tfvars`) if present in the current directory.

7. Copy the following code into your variables file. Make sure to replace the placeholders as follows: For `subscription_id`, use the Azure subscription ID you specified when running `az account set`. For `tenant_id`, use the `tenant` value returned from `az ad sp create-for-rbac`. For `client_id`, use the `appId` value returned from `az ad sp create-for-rbac`. For `client_secret`, use the `password` value returned from `az ad sp create-for-rbac`.

```
subscription_id = "<azure-subscription-id>"
tenant_id = "<tenant-returned-from-creating-a-service-principal>"
client_id = "<appId-returned-from-creating-a-service-principal>"
client_secret = "<password-returned-from-creating-a-service-principal>"
```

## 2. Create a Terraform configuration file

In this section, you create a file that contains resource definitions for your infrastructure.

1. Create a new file named `main.tf`.
2. Copy following sample resource definitions into the newly created `main.tf` file:

```
resource "azurerm_resource_group" "test" {
  name     = "acctestrg"
  location = "West US 2"
}

resource "azurerm_virtual_network" "test" {
  name            = "acctvn"
  address_space   = ["10.0.0.0/16"]
  location        = azurerm_resource_group.test.location
  resource_group_name = azurerm_resource_group.test.name
}

resource "azurerm_subnet" "test" {
  name                 = "acctsub"
  resource_group_name = azurerm_resource_group.test.name
  virtual_network_name = azurerm_virtual_network.test.name
  address_prefix       = "10.0.2.0/24"
}

resource "azurerm_public_ip" "test" {
  name                 = "publicIPForLB"
  location             = azurerm_resource_group.test.location
  resource_group_name = azurerm_resource_group.test.name
  allocation_method    = "Static"
}

resource "azurerm_lb" "test" {
  name                 = "loadBalancer"
  location             = azurerm_resource_group.test.location
  resource_group_name = azurerm_resource_group.test.name

  frontend_ip_configuration {
    name                 = "publicIPAddress"
    public_ip_address_id = azurerm_public_ip.test.id
  }
}

resource "azurerm_lb_backend_address_pool" "test" {
  resource_group_name = azurerm_resource_group.test.name
  loadbalancer_id     = azurerm_lb.test.id
  name                = "BackEndAddressPool"
}

resource "azurerm_network_interface" "test" {
```

```

count          = 2
name           = "acctni${count.index}"
location       = azurerm_resource_group.test.location
resource_group_name = azurerm_resource_group.test.name

ip_configuration {
  name              = "testConfiguration"
  subnet_id        = azurerm_subnet.test.id
  private_ip_address_allocation = "dynamic"
}
}

resource "azurerm_managed_disk" "test" {
  count          = 2
  name           = "datadisk_existing_${count.index}"
  location       = azurerm_resource_group.test.location
  resource_group_name = azurerm_resource_group.test.name
  storage_account_type = "Standard_LRS"
  create_option   = "Empty"
  disk_size_gb    = "1023"
}

resource "azurerm_availability_set" "avset" {
  name              = "avset"
  location          = azurerm_resource_group.test.location
  resource_group_name = azurerm_resource_group.test.name
  platform_fault_domain_count = 2
  platform_update_domain_count = 2
  managed           = true
}

resource "azurerm_virtual_machine" "test" {
  count          = 2
  name           = "acctvm${count.index}"
  location       = azurerm_resource_group.test.location
  availability_set_id = azurerm_availability_set.avset.id
  resource_group_name = azurerm_resource_group.test.name
  network_interface_ids = [element(azurerm_network_interface.test.*.id, count.index)]
  vm_size        = "Standard_DS1_v2"

  # Uncomment this line to delete the OS disk automatically when deleting the VM
  # delete_os_disk_on_termination = true

  # Uncomment this line to delete the data disks automatically when deleting the VM
  # delete_data_disks_on_termination = true

  storage_image_reference {
    publisher = "Canonical"
    offer     = "UbuntuServer"
    sku       = "16.04-LTS"
    version   = "latest"
  }

  storage_os_disk {
    name              = "myosdisk${count.index}"
    caching           = "ReadWrite"
    create_option     = "FromImage"
    managed_disk_type = "Standard_LRS"
  }

  # Optional data disks
  storage_data_disk {
    name              = "datadisk_new_${count.index}"
    managed_disk_type = "Standard_LRS"
    create_option     = "Empty"
    lun               = 0
    disk_size_gb      = "1023"
  }
}

```

```
storage_data_disk {
  name           = element(azurerm_managed_disk.test.*.name, count.index)
  managed_disk_id = element(azurerm_managed_disk.test.*.id, count.index)
  create_option   = "Attach"
  lun            = 1
  disk_size_gb    = element(azurerm_managed_disk.test.*.disk_size_gb, count.index)
}

os_profile {
  computer_name  = "hostname"
  admin_username = "testadmin"
  admin_password = "Password1234!"
}

os_profile_linux_config {
  disable_password_authentication = false
}

tags = {
  environment = "staging"
}
```

### 3. Initialize Terraform

The `terraform init` command is used to initialize a directory that contains the Terraform configuration files - the files you created with the previous sections. It's a good practice to always run the `terraform init` command after writing a new Terraform configuration.

#### TIP

The `terraform init` command is idempotent meaning that it can be called repeatedly while producing the same result. Therefore, if you're working in a collaborative environment, and you think the configuration files might have been changed, it's always a good idea to call the `terraform init` command before executing or applying a plan.

To initialize Terraform, run the following command:

```
terraform init
```

```
C:\terraform-test>terraform init

Initializing provider plugins...
- Checking for available provider plugins on https://releases.hashicorp.com...
- Downloading plugin for provider "azurerm" (0.3.2)...

The following providers do not have any version constraints in configuration,
so the latest version was installed.

To prevent automatic upgrades to new major versions that may contain breaking
changes, it is recommended to add version = "..." constraints to the
corresponding provider blocks in configuration, with the constraint strings
suggested below.

* provider.azurerm: version = "~> 0.3"

Terraform has been successfully initialized!

You may now begin working with Terraform. Try running "terraform plan" to see
any changes that are required for your infrastructure. All Terraform commands
should now work.

If you ever set or change modules or backend configuration for Terraform,
rerun this command to reinitialize your working directory. If you forget, other
commands will detect it and remind you to do so if necessary.

C:\terraform-test>
```

## 4. Create a Terraform execution plan

The [terraform plan command](#) is used to create an execution plan. To generate an execution plan, Terraform aggregates all the `.tf` files in the current directory.

The [-out parameter](#) saves the execution plan to an output file. This feature addresses concurrency issues common in multi-dev environments. One such problem solved by the output file is the following scenario:

1. Dev 1 creates the configuration file.
2. Dev 2 modifies the configuration file.
3. Dev 1 applies (runs) the configuration file.
4. Dev 1 gets unexpected results not knowing that Dev 2 modified the configuration.

Dev 1 specifying an output file prevents Dev 2 from affecting Dev 1.

If you don't need to save your execution plan, run the following command:

```
terraform plan
```

If you need to save your execution plan, run the following command. Replace the placeholders with appropriate values for your environment.

```
terraform plan -out=<path>
```

Another useful parameter is [-var-file](#).

By default Terraform tried to find your variables file as follows:

- File named `terraform.tfvars`
- File named with using the following pattern: `*.auto.tfvars`

However, your variables file need not follow either of the two preceding conventions. In that case, specify your variables file name with the `-var-file` parameter where your variable file name does not carry an extension. The following example illustrates this point:

```
terraform plan -var-file <my-variables-file>
```

Terraform determines the actions necessary to achieve the state specified in the configuration file.

```
C:\terraform-test>terraform plan
Refreshing Terraform state in-memory prior to plan...
The refreshed state will be used to calculate this plan, but will not be
persisted to local or remote state storage.

-----

An execution plan has been generated and is shown below.
Resource actions are indicated with the following symbols:
+ create

Terraform will perform the following actions:

+ azurerm_availability_set.avset
  id:                                <computed>
  location:                          "westus2"
  managed:                           "true"
  name:                              "avset"
  platform_fault_domain_count:       "2"
  platform_update_domain_count:      "2"
  resource_group_name:               "acctestrg"
  tags.%:                            <computed>

+ azurerm_lb.test
  id:                                <computed>
```

## 5. Apply the Terraform execution plan

The final step of this tutorial is to use the [terraform apply command](#) to apply the set of actions generated by the `terraform plan` command.

If you want to apply the latest execution plan, run the following command:

```
terraform apply
```

If you want to apply a previously saved execution plan, run the following command. Replace the placeholders with appropriate values for your environment:

```
terraform apply <path>
```

```

C:\terraform-test>terraform apply
azurerm_resource_group.test: Creating...
  location: "" => "westus2"
  name:      "" => "acctestrg"
  tags.%:    "" => "<computed>"
azurerm_resource_group.test: Creation complete after 0s (ID: /subscriptions/ad7af
azurerm_managed_disk.test[1]: Creating...
  create_option: "" => "Empty"
  disk_size_gb:  "" => "1023"
  location:      "" => "westus2"
  name:          "" => "datadisk_existing_1"
  resource_group_name: "" => "acctestrg"
  source_uri:      "" => "<computed>"
  storage_account_type: "" => "Standard_LRS"
  tags.%:         "" => "<computed>"
azurerm_virtual_network.test: Creating...
  address_space.#: "" => "1"
  address_space.0: "" => "10.0.0.0/16"
  location:        "" => "westus2"
  name:            "" => "acctvn"
  resource_group_name: "" => "acctestrg"
  subnet.#:        "" => "<computed>"
  tags.%:          "" => "<computed>"
azurerm_availability_set.avset: Creating...
  location: "" => "westus2"
  managed:  "" => "true"

```

## Next steps

[Create an Azure virtual machine scale set using Terraform](#)

minutes to read • [Edit Online](#)

[Azure virtual machine scale sets](#) allow you to configure identical VMs. The number of VM instances can adjust based on demand or a schedule. For more information, see [Automatically scale a virtual machine scale set in the Azure portal](#).

In this tutorial, you learn how to use [Azure Cloud Shell](#) to do the following tasks:

- Set up a Terraform deployment
- Use variables and outputs for Terraform deployment
- Create and deploy network infrastructure
- Create and deploy a virtual machine scale set and attach it to the network
- Create and deploy a jumpbox to connect to the VMs via SSH

#### NOTE

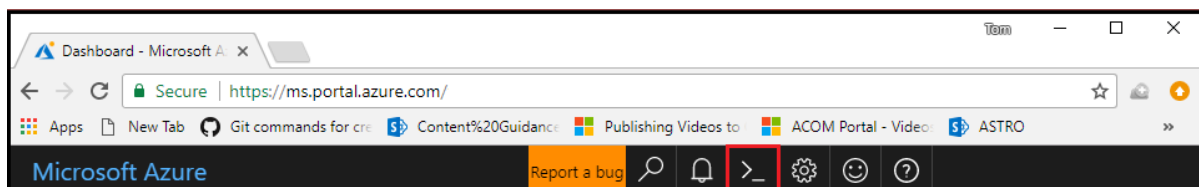
The most recent version of the Terraform configuration files used in this article are in the [Awesome Terraform repository on GitHub](#).

## Prerequisites

- **Azure subscription:** If you don't have an Azure subscription, create a [free account](#) before you begin.
- **Install Terraform:** Follow the directions in the article, [Terraform and configure access to Azure](#)
- **Create an SSH key pair:** For more information, see [How to create and use an SSH public and private key pair for Linux VMs in Azure](#).

## Create the directory structure

1. Browse to the [Azure portal](#).
2. Open [Azure Cloud Shell](#). If you didn't select an environment previously, select **Bash** as your environment.



3. Change directories to the `clouddrive` directory.

```
cd clouddrive
```

4. Create a directory named `vmss`.

```
mkdir vmss
```

5. Change directories to the new directory:



```
cd vmss
```

## Create the variables definitions file

In this section, you define the variables that customize the resources created by Terraform.

Within the Azure Cloud Shell, do the following steps:

1. Create a file named `variables.tf`.

```
code variables.tf
```

2. Paste the following code into the editor:

```
variable "location" {
  description = "The location where resources will be created"
}

variable "tags" {
  description = "A map of the tags to use for the resources that are deployed"
  type        = "map"

  default = {
    environment = "codelab"
  }
}

variable "resource_group_name" {
  description = "The name of the resource group in which the resources will be created"
  default     = "myResourceGroup"
}
```

3. Save the file (<Ctrl>S) and exit the editor (<Ctrl>Q).

## Create the output definitions file

In this section, you create the file that describes the output after deployment.

Within the Azure Cloud Shell, do the following steps:

1. Create a file named `output.tf`.

```
code output.tf
```

2. Paste the following code into the editor to expose the fully qualified domain name (FQDN) for the virtual machines.:

```
output "vmss_public_ip" {
  value = azurerm_public_ip.vmss.fqdn
}
```

3. Save the file (<Ctrl>S) and exit the editor (<Ctrl>Q).

## Define the network infrastructure in a template

In this section, you create the following network infrastructure in a new Azure resource group:

- One virtual network (VNET) with the address space of 10.0.0.0/16
- One subnet with the address space of 10.0.2.0/24
- Two public IP addresses. One used by the virtual machine scale set load balancer, the other used to connect to the SSH jumpbox.

Within the Azure Cloud Shell, do the following steps:

1. Create a file named `vmss.tf` to describe the virtual machine scale set infrastructure.

```
code vmss.tf
```

2. Paste the following code to the end of the file to expose the fully qualified domain name (FQDN) for the virtual machines.

```
resource "azurerm_resource_group" "vmss" {
  name      = var.resource_group_name
  location  = var.location
  tags      = var.tags
}

resource "random_string" "fqdn" {
  length    = 6
  special   = false
  upper     = false
  number    = false
}

resource "azurerm_virtual_network" "vmss" {
  name                = "vmss-vnet"
  address_space       = ["10.0.0.0/16"]
  location            = var.location
  resource_group_name = azurerm_resource_group.vmss.name
  tags                = var.tags
}

resource "azurerm_subnet" "vmss" {
  name                 = "vmss-subnet"
  resource_group_name  = azurerm_resource_group.vmss.name
  virtual_network_name = azurerm_virtual_network.vmss.name
  address_prefix        = "10.0.2.0/24"
}

resource "azurerm_public_ip" "vmss" {
  name                = "vmss-public-ip"
  location            = var.location
  resource_group_name = azurerm_resource_group.vmss.name
  allocation_method   = "Static"
  domain_name_label    = random_string.fqdn.result
  tags                = var.tags
}
```

3. Save the file (<Ctrl>S) and exit the editor (<Ctrl>Q).

## Provision the network infrastructure

Using the Azure Cloud Shell from the directory where you created the configuration files (.tf) do the following steps:

1. Initialize Terraform.

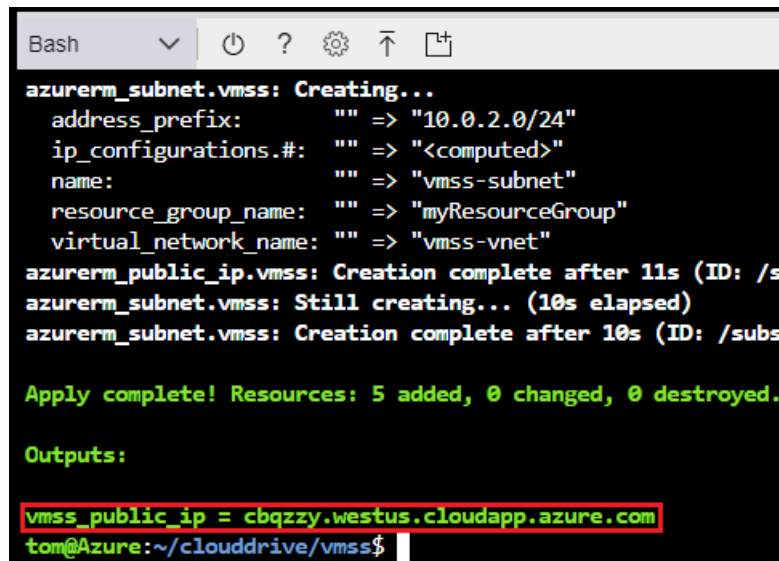
```
terraform init
```

2. Run the following command to deploy the defined infrastructure in Azure.

```
terraform apply
```

Terraform prompts you for a `location` value as the `location` variable is defined in `variables.tf`, but it's never set. You can enter any valid location - such as "West US" followed by selecting Enter. (Use parentheses around any value with spaces.)

3. Terraform prints the output as defined in the `output.tf` file. As shown in the following screenshot, the FQDN takes the following form: `<ID>.<location>.cloudapp.azure.com`. The ID is a computed value and location is the value provide when running Terraform.

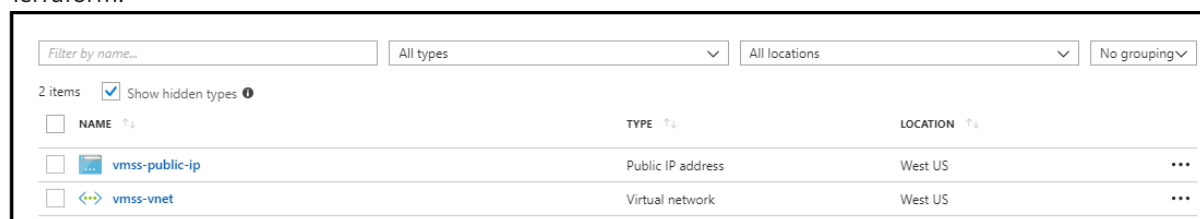


```
Bash
azurerm_subnet.vmss: Creating...
  address_prefix: "" => "10.0.2.0/24"
  ip_configurations.#: "" => "<computed>"
  name: "" => "vmss-subnet"
  resource_group_name: "" => "myResourceGroup"
  virtual_network_name: "" => "vmss-vnet"
azurerm_public_ip.vmss: Creation complete after 11s (ID: /s
azurerm_subnet.vmss: Still creating... (10s elapsed)
azurerm_subnet.vmss: Creation complete after 10s (ID: /subs
Apply complete! Resources: 5 added, 0 changed, 0 destroyed.

Outputs:

vmss_public_ip = cbqzzy.westus.cloudapp.azure.com
tom@Azure:~/clouddrive/vmss$
```

4. In the Azure portal menu, select Resource groups from the main menu.
5. On the Resource groups tab, select myResourceGroup to view the resources that were created by Terraform.



NAME	TYPE	LOCATION
vmss-public-ip	Public IP address	West US
vmss-vnet	Virtual network	West US

## Add a virtual machine scale set

In this section, you learn how to add the following resources to the template:

- An Azure load balancer and rules to serve the application and attach it to the public IP address configured earlier in this article
- An Azure backend address pool and assign it to the load balancer
- A health probe port used by the application and configured on the load balancer
- A virtual machine scale set sitting behind the load balancer that runs on the VNET deployed earlier in this article
- [Nginx](#) on the nodes of the virtual machine scale using [cloud-init](#).

In Cloud Shell, do the following steps:

1. Open the `vmss.tf` configuration file.

```
code vmss.tf
```

2. Go to the end of the file and enter append mode by selecting the A key.
3. Paste the following code to the end of the file:

```
resource "azurerm_lb" "vmss" {
  name                = "vmss-lb"
  location            = var.location
  resource_group_name = azurerm_resource_group.vmss.name

  frontend_ip_configuration {
    name                = "PublicIPAddress"
    public_ip_address_id = azurerm_public_ip.vmss.id
  }

  tags = var.tags
}

resource "azurerm_lb_backend_address_pool" "bpepool" {
  resource_group_name = azurerm_resource_group.vmss.name
  loadbalancer_id      = azurerm_lb.vmss.id
  name                 = "BackEndAddressPool"
}

resource "azurerm_lb_probe" "vmss" {
  resource_group_name = azurerm_resource_group.vmss.name
  loadbalancer_id      = azurerm_lb.vmss.id
  name                 = "ssh-running-probe"
  port                 = var.application_port
}

resource "azurerm_lb_rule" "lbnatrule" {
  resource_group_name      = azurerm_resource_group.vmss.name
  loadbalancer_id          = azurerm_lb.vmss.id
  name                     = "http"
  protocol                 = "Tcp"
  frontend_port            = var.application_port
  backend_port             = var.application_port
  backend_address_pool_id  = azurerm_lb_backend_address_pool.bpepool.id
  frontend_ip_configuration_name = "PublicIPAddress"
  probe_id                 = azurerm_lb_probe.vmss.id
}

resource "azurerm_virtual_machine_scale_set" "vmss" {
  name                = "vmscaleset"
  location            = var.location
  resource_group_name = azurerm_resource_group.vmss.name
  upgrade_policy_mode = "Manual"

  sku {
    name     = "Standard_DS1_v2"
    tier      = "Standard"
    capacity = 2
  }

  storage_profile_image_reference {
    publisher = "Canonical"
    offer     = "UbuntuServer"
    sku       = "16.04-LTS"
    version   = "latest"
  }

  storage_profile_os_disk {
```

```

    name          = ""
    caching        = "ReadWrite"
    create_option  = "FromImage"
    managed_disk_type = "Standard_LRS"
  }

  storage_profile_data_disk {
    lun          = 0
    caching      = "ReadWrite"
    create_option = "Empty"
    disk_size_gb = 10
  }

  os_profile {
    computer_name_prefix = "vmlab"
    admin_username       = var.admin_user
    admin_password       = var.admin_password
    custom_data          = file("web.conf")
  }

  os_profile_linux_config {
    disable_password_authentication = false
  }

  network_profile {
    name = "terraformnetworkprofile"
    primary = true

    ip_configuration {
      name                          = "IPConfiguration"
      subnet_id                    = azurerm_subnet.vmss.id
      load_balancer_backend_address_pool_ids = [azurerm_lb_backend_address_pool.bpepool.id]
      primary = true
    }
  }

  tags = var.tags
}

```

4. Save the file and exit the vi editor by entering the following command:

```
:wq
```

5. Create a file named `web.conf` to serve as the cloud-init configuration for the virtual machines that are part of the scale set.

```
code web.conf
```

6. Paste the following code into the editor:

```
#cloud-config
packages:
- nginx
```

7. Save the file and exit the vi editor by entering the following command:

```
:wq
```

8. Open the `variables.tf` configuration file.

```
code variables.tf
```

9. Go to the end of the file and enter append mode by selecting the A key.
10. Customize the deployment by pasting the following code to the end of the file:

```
variable "application_port" {
  description = "The port that you want to expose to the external load balancer"
  default     = 80
}

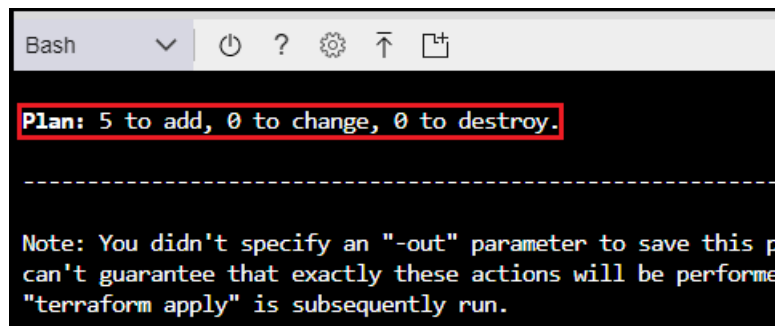
variable "admin_user" {
  description = "User name to use as the admin account on the VMs that will be part of the VM Scale Set"
  default     = "azureuser"
}

variable "admin_password" {
  description = "Default password for admin account"
}
```

11. Save the file (<Ctrl>S) and exit the editor (<Ctrl>Q).
12. Create a Terraform plan to visualize the virtual machine scale set deployment. (You need to specify a password of your choosing, as well as the location for your resources.)

```
terraform plan
```

The output of the command should be similar to the following screenshot:



```
Bash
Plan: 5 to add, 0 to change, 0 to destroy.

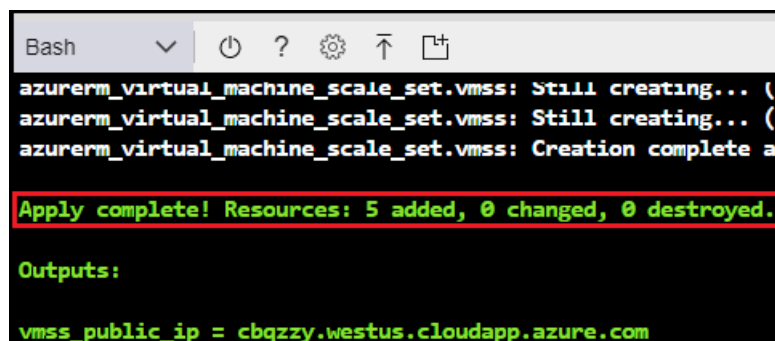
-----

Note: You didn't specify an "-out" parameter to save this plan. Terraform
can't guarantee that exactly these actions will be performed if the
"terraform apply" is subsequently run.
```

13. Deploy the new resources in Azure.

```
terraform apply
```

The output of the command should be similar to the following screenshot:

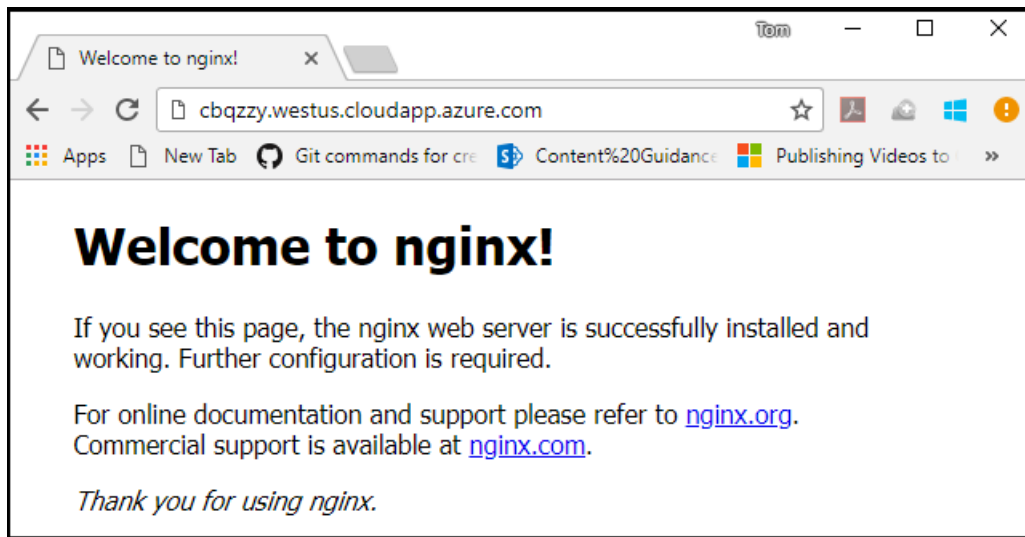


```
Bash
azurerm_virtual_machine_scale_set.vmss: Still creating... (1s)
azurerm_virtual_machine_scale_set.vmss: Still creating... (2s)
azurerm_virtual_machine_scale_set.vmss: Creation complete at 2023-01-10 14:00:00 UTC

Apply complete! Resources: 5 added, 0 changed, 0 destroyed.

Outputs:
vmss_public_ip = cbqzzy.westus.cloudapp.azure.com
```

14. Open a browser and connect to the FQDN that was returned by the command.



## Add an SSH jumpbox

An SSH *jumpbox* is a single server that you "jump" through to access other servers on the network. In this step, you configure the following resources:

- A network interface (or jumpbox) connected to the same subnet as the virtual machine scale set.
- A virtual machine connected with this network interface. This 'jumpbox' is remotely accessible. Once connected, you can SSH to any of the virtual machines in the scale set.

1. Open the `vmss.tf` configuration file.

```
code vmss.tf
```

2. Go to the end of the file and enter append mode by selecting the A key.

3. Paste the following code to the end of the file:

```

resource "azurerm_public_ip" "jumpbox" {
  name                = "jumpbox-public-ip"
  location             = var.location
  resource_group_name = azurerm_resource_group.vmss.name
  allocation_method    = "Static"
  domain_name_label    = "${random_string.fqdn.result}-ssh"
  tags                = var.tags
}

resource "azurerm_network_interface" "jumpbox" {
  name                = "jumpbox-nic"
  location            = var.location
  resource_group_name = azurerm_resource_group.vmss.name

  ip_configuration {
    name                       = "IPConfiguration"
    subnet_id                 = azurerm_subnet.vmss.id
    private_ip_address_allocation = "dynamic"
    public_ip_address_id       = azurerm_public_ip.jumpbox.id
  }

  tags = var.tags
}

resource "azurerm_virtual_machine" "jumpbox" {
  name                = "jumpbox"
  location            = var.location
  resource_group_name = azurerm_resource_group.vmss.name
  network_interface_ids = [azurerm_network_interface.jumpbox.id]
  vm_size             = "Standard_DS1_v2"

  storage_image_reference {
    publisher = "Canonical"
    offer     = "UbuntuServer"
    sku       = "16.04-LTS"
    version   = "latest"
  }

  storage_os_disk {
    name            = "jumpbox-osdisk"
    caching         = "ReadWrite"
    create_option   = "FromImage"
    managed_disk_type = "Standard_LRS"
  }

  os_profile {
    computer_name  = "jumpbox"
    admin_username = var.admin_user
    admin_password = var.admin_password
  }

  os_profile_linux_config {
    disable_password_authentication = false
  }

  tags = var.tags
}

```

4. Open the `output.tf` configuration file.

```
code output.tf
```

5. Go to the end of the file and enter append mode by selecting the A key.
6. Paste the following code to the end of the file to display the hostname of the jumpbox when the deployment



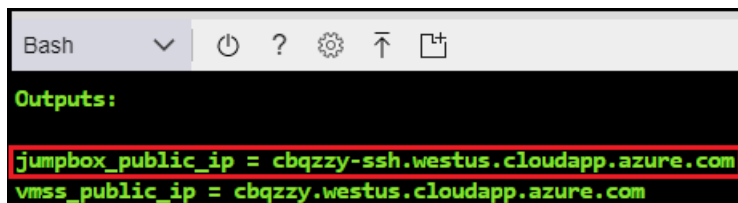
is complete:

```
output "jumpbox_public_ip" {  
  value = azurermp_public_ip.jumpbox.fqdn  
}
```

7. Save the file (<Ctrl>S) and exit the editor (<Ctrl>Q).
8. Deploy the jumpbox.

```
terraform apply
```

Once the deployment has completed, the content of the resource group resembles that shown in the following screenshot:



#### NOTE

The ability to log in with a password is disabled on the jumpbox and the virtual machine scale set that you deployed. Log in with SSH to access the virtual machine(s).

## Environment cleanup

To delete the Terraform resources that were created in this tutorial, enter the following command into Cloud Shell:

```
terraform destroy
```

The destruction process can take several minutes to complete.

## Next steps

[Learn more about using Terraform in Azure](#)

In this tutorial, you use [Terraform](#) to create and deploy an [Azure virtual machine scale set](#) created with a custom image produced by using [Packer](#) with managed disks that use the [HashiCorp Configuration Language](#) (HCL).

In this tutorial, you learn how to:

- Set up your Terraform deployment.
- Use variables and outputs for Terraform deployment.
- Create and deploy a network infrastructure.
- Create a custom virtual machine image by using Packer.
- Create and deploy a virtual machine scale set by using the custom image.
- Create and deploy a jumpbox.

If you don't have an Azure subscription, create a [free account](#) before you begin.

## Prerequisites

- **Terraform:** [Install Terraform and configure access to Azure](#).
- **SSH key pair:** [Create an SSH key pair](#).
- **Packer:** [Install Packer](#).

## Create the file structure

Create three new files in an empty directory with the following names:

- `variables.tf`: This file holds the values of the variables used in the template.
- `output.tf`: This file describes the settings that display after deployment.
- `vmss.tf`: This file contains the code of the infrastructure that you're deploying.

## Create the variables

In this step, you define variables that customize the resources created by Terraform.

Edit the `variables.tf` file, copy the following code, and then save the changes.

```
variable "location" {
  description = "The location where resources are created"
  default     = "East US"
}

variable "resource_group_name" {
  description = "The name of the resource group in which the resources are created"
  default     = ""
}
```

### NOTE

The default value of the `resource_group_name` variable is unset. Define your own value.

Save the file.

When you deploy your Terraform template, you want to get the fully qualified domain name that's used to access the application. Use the `output` resource type of Terraform, and get the `fqdn` property of the resource.

Edit the `output.tf` file, and copy the following code to expose the fully qualified domain name for the virtual machines.

```
output "vmss_public_ip" {  
  value = azurerm_public_ip.vmss.fqdn  
}
```

## Define the network infrastructure in a template

In this step, you create the following network infrastructure in a new Azure resource group:

- One virtual network with the address space of 10.0.0.0/16.
- One subnet with the address space of 10.0.2.0/24.
- Two public IP addresses. One is used by the virtual machine scale set load balancer. The other is used to connect to the SSH jumpbox.

You also need a resource group where all the resources are created.

Edit and copy the following code in the `vmss.tf` file:

```

resource "azurerm_resource_group" "vmss" {
  name      = var.resource_group_name
  location  = var.location

  tags {
    environment = "codelab"
  }
}

resource "azurerm_virtual_network" "vmss" {
  name            = "vmss-vnet"
  address_space   = ["10.0.0.0/16"]
  location        = var.location
  resource_group_name = azurerm_resource_group.vmss.name

  tags {
    environment = "codelab"
  }
}

resource "azurerm_subnet" "vmss" {
  name                 = "vmss-subnet"
  resource_group_name = azurerm_resource_group.vmss.name
  virtual_network_name = azurerm_virtual_network.vmss.name
  address_prefix       = "10.0.2.0/24"
}

resource "azurerm_public_ip" "vmss" {
  name                 = "vmss-public-ip"
  location             = var.location
  resource_group_name = azurerm_resource_group.vmss.name
  allocation_method    = "static"
  domain_name_label    = azurerm_resource_group.vmss.name

  tags {
    environment = "codelab"
  }
}

```

#### NOTE

Tag the resources that are being deployed in Azure to facilitate their identification in the future.

## Create the network infrastructure

Initialize the Terraform environment by running the following command in the directory where you created the `.tf` files:

```
terraform init
```

The provider plug-ins download from the Terraform registry into the `.terraform` folder in the directory where you ran the command.

Run the following command to deploy the infrastructure in Azure.

```
terraform apply
```

Verify that the fully qualified domain name of the public IP address corresponds to your configuration.

## Outputs:

```
vmss_public_ip = tfdocsvmss.westus.cloudapp.azure.com
Damien-MacBook-Pro:fulltest dcaro$
```

The resource group contains the following resources:

	 <b>jumpbox-public-ip</b>	Public IP address	West US	...
	 <b>vmss-public-ip</b>	Public IP address	West US	...
	 <b>vmss-vnet</b>	Virtual network	West US	...

## Create an Azure image by using Packer

Create a custom Linux image by following the steps in the tutorial [How to use Packer to create Linux virtual machine images in Azure](#).

Follow the tutorial to create a deprovisioned Ubuntu image with Nginx installed.

```
==> Builds finished. The artifacts of successful builds are:
--> azure-arm: Azure.ResourceManagement.VMImage:

ManagedImageResourceGroupName: myResourceGroup
ManagedImageName: myPackerImage
ManagedImageLocation: eastus
```

### NOTE

For the purposes of this tutorial, in the Packer image, a command is run to install Nginx. You can also run your own script while creating.

## Edit the infrastructure to add the virtual machine scale set

In this step, you create the following resources on the network that was previously deployed:

- An Azure load balancer to serve the application. Attach it to the public IP address that was deployed earlier.
- One Azure load balancer and rules to serve the application. Attach it to the public IP address that was configured earlier.
- An Azure back-end address pool. Assign it to the load balancer.
- A health probe port used by the application and configured on the load balancer.
- A virtual machine scale set that sits behind the load balancer and runs on the virtual network that was deployed earlier.
- [Nginx](#) on the nodes of the virtual machine scale installed from a custom image.

Add the following code to the end of the `vmss.tf` file.

```
resource "azurerm_lb" "vmss" {
  name = "vmss-lb"
```

```

location          = var.location
resource_group_name = azurerm_resource_group.vmss.name

frontend_ip_configuration {
  name          = "PublicIPAddress"
  public_ip_address_id = azurerm_public_ip.vmss.id
}

tags {
  environment = "codelab"
}
}

resource "azurerm_lb_backend_address_pool" "bpepool" {
  resource_group_name = azurerm_resource_group.vmss.name
  loadbalancer_id      = azurerm_lb.vmss.id
  name                 = "BackEndAddressPool"
}

resource "azurerm_lb_probe" "vmss" {
  resource_group_name = azurerm_resource_group.vmss.name
  loadbalancer_id      = azurerm_lb.vmss.id
  name                 = "ssh-running-probe"
  port                 = var.application_port
}

resource "azurerm_lb_rule" "lbnatrule" {
  resource_group_name      = azurerm_resource_group.vmss.name
  loadbalancer_id          = azurerm_lb.vmss.id
  name                     = "http"
  protocol                  = "Tcp"
  frontend_port             = var.application_port
  backend_port              = var.application_port
  backend_address_pool_id   = azurerm_lb_backend_address_pool.bpepool.id
  frontend_ip_configuration_name = "PublicIPAddress"
  probe_id                 = azurerm_lb_probe.vmss.id
}

data "azurerm_resource_group" "image" {
  name = "myResourceGroup"
}

data "azurerm_image" "image" {
  name          = "myPackerImage"
  resource_group_name = data.azurerm_resource_group.image.name
}

resource "azurerm_virtual_machine_scale_set" "vmss" {
  name          = "vmscaleset"
  location      = var.location
  resource_group_name = azurerm_resource_group.vmss.name
  upgrade_policy_mode = "Manual"

  sku {
    name     = "Standard_DS1_v2"
    tier      = "Standard"
    capacity = 2
  }

  storage_profile_image_reference {
    id = data.azurerm_image.image.id
  }

  storage_profile_os_disk {
    name          = ""
    caching        = "ReadWrite"
    create_option = "FromImage"
    managed_disk_type = "Standard_LRS"
  }
}

```

```

storage_profile_data_disk {
  lun          = 0
  caching      = "ReadWrite"
  create_option = "Empty"
  disk_size_gb = 10
}

os_profile {
  computer_name_prefix = "vmlab"
  admin_username       = "azureuser"
  admin_password       = "Password1234"
}

os_profile_linux_config {
  disable_password_authentication = true

  ssh_keys {
    path      = "/home/azureuser/.ssh/authorized_keys"
    key_data = file("~/ssh/id_rsa.pub")
  }
}

network_profile {
  name      = "terraformnetworkprofile"
  primary   = true

  ip_configuration {
    name                        = "IPConfiguration"
    subnet_id                  = azurerm_subnet.vmss.id
    load_balancer_backend_address_pool_ids = [azurerm_lb_backend_address_pool.bpepool.id]
    primary = true
  }
}

tags {
  environment = "codelab"
}
}

```

Customize the deployment by adding the following code to `variables.tf`:

```

variable "application_port" {
  description = "The port that you want to expose to the external load balancer"
  default     = 80
}

variable "admin_password" {
  description = "Default password for admin"
  default     = "Passwwoord11223344"
}

```

## Deploy the virtual machine scale set in Azure

Run the following command to visualize the virtual machine scale set deployment:

```
terraform plan
```

The output of the command looks like the following image:

**Plan: 5 to add, 0 to change, 0 to destroy.**

---

**Note:** You didn't specify an "-out" parameter to save this plan, so Terraform can't guarantee that exactly these actions will be performed if "terraform apply" is subsequently run.

Deploy the additional resources in Azure:

```
terraform apply
```

The content of the resource group looks like the following image:

```
azurerm_virtual_machine_scale_set.vmss: Still creating... (2m50s elapsed)
azurerm_virtual_machine_scale_set.vmss: Still creating... (3m0s elapsed)
azurerm_virtual_machine_scale_set.vmss: Creation complete after 3m3s (ID: /subscriptions/.../resourceGroups/.../providers/Microsoft.Compute/virtualMachineScaleSets/vmss)

Apply complete! Resources: 5 added, 0 changed, 0 destroyed.
```

Open a browser and connect to the fully qualified domain name that was returned by the command.

## Add a jumpbox to the existing network

This optional step enables SSH access to the instances of the virtual machine scale set by using a jumpbox.

Add the following resources to your existing deployment:

- A network interface connected to the same subnet as the virtual machine scale set
- A virtual machine with this network interface

Add the following code to the end of the `vmss.tf` file:

```
resource "azurerm_public_ip" "jumpbox" {
  name                = "jumpbox-public-ip"
  location            = var.location
  resource_group_name = azurerm_resource_group.vmss.name
  allocation_method   = "static"
  domain_name_label   = "${azurerm_resource_group.vmss.name}-ssh"

  tags {
    environment = "codelab"
  }
}

resource "azurerm_network_interface" "jumpbox" {
  name                = "jumpbox-nic"
  location            = var.location
  resource_group_name = azurerm_resource_group.vmss.name

  ip_configuration {
    name                          = "IPConfiguration"
    subnet_id                    = azurerm_subnet.vmss.id
    private_ip_address_allocation = "dynamic"
    public_ip_address_id         = azurerm_public_ip.jumpbox.id
  }

  tags {
    environment = "codelab"
  }
}
```



```

resource "azurerm_virtual_machine" "jumpbox" {
  name                = "jumpbox"
  location            = var.location
  resource_group_name = azurerm_resource_group.vmss.name
  network_interface_ids = [azurerm_network_interface.jumpbox.id]
  vm_size             = "Standard_DS1_v2"

  storage_image_reference {
    publisher = "Canonical"
    offer     = "UbuntuServer"
    sku       = "16.04-LTS"
    version   = "latest"
  }

  storage_os_disk {
    name          = "jumpbox-osdisk"
    caching       = "ReadWrite"
    create_option = "FromImage"
    managed_disk_type = "Standard_LRS"
  }

  os_profile {
    computer_name  = "jumpbox"
    admin_username = "azureuser"
    admin_password = "Password1234!"
  }

  os_profile_linux_config {
    disable_password_authentication = true

    ssh_keys {
      path      = "/home/azureuser/.ssh/authorized_keys"
      key_data = file("~/ssh/id_rsa.pub")
    }
  }

  tags {
    environment = "codelab"
  }
}

```

Edit `outputs.tf` to add the following code that displays the hostname of the jumpbox when the deployment finishes:

```

output "jumpbox_public_ip" {
  value = azurerm_public_ip.jumpbox.fqdn
}

```

## Deploy the jumpbox

Deploy the jumpbox.

```
terraform apply
```

After the deployment has completed, the content of the resource group looks like the following image:

## Outputs:

```
jumpbox_public_ip = codelab-ssh.westus.cloudapp.azure.com
vmss_public_ip    = codelab.westus.cloudapp.azure.com
```

### NOTE

Sign-in with a password is disabled on the jumpbox and the virtual machine scale set that you deployed. Sign in with SSH to access the VMs.

## Clean up the environment

The following commands delete the resources created in this tutorial:

```
terraform destroy
```

Enter *yes* when you're asked to confirm the deletion of the resources. The destruction process can take a few minutes to complete.

## Next steps

[Learn more about using Terraform in Azure](#)

This tutorial series shows how to use Terraform to implement in Azure a [hub and spoke network topology](#).

A hub and spoke topology is a way to isolate workloads while sharing common services. These services include identity and security. The hub is a virtual network (VNet) that acts as a central connection point to an on-premises network. The spokes are VNets that peer with the hub. Shared services are deployed in the hub, while individual workloads are deployed inside spoke networks.

This tutorial covers the following tasks:

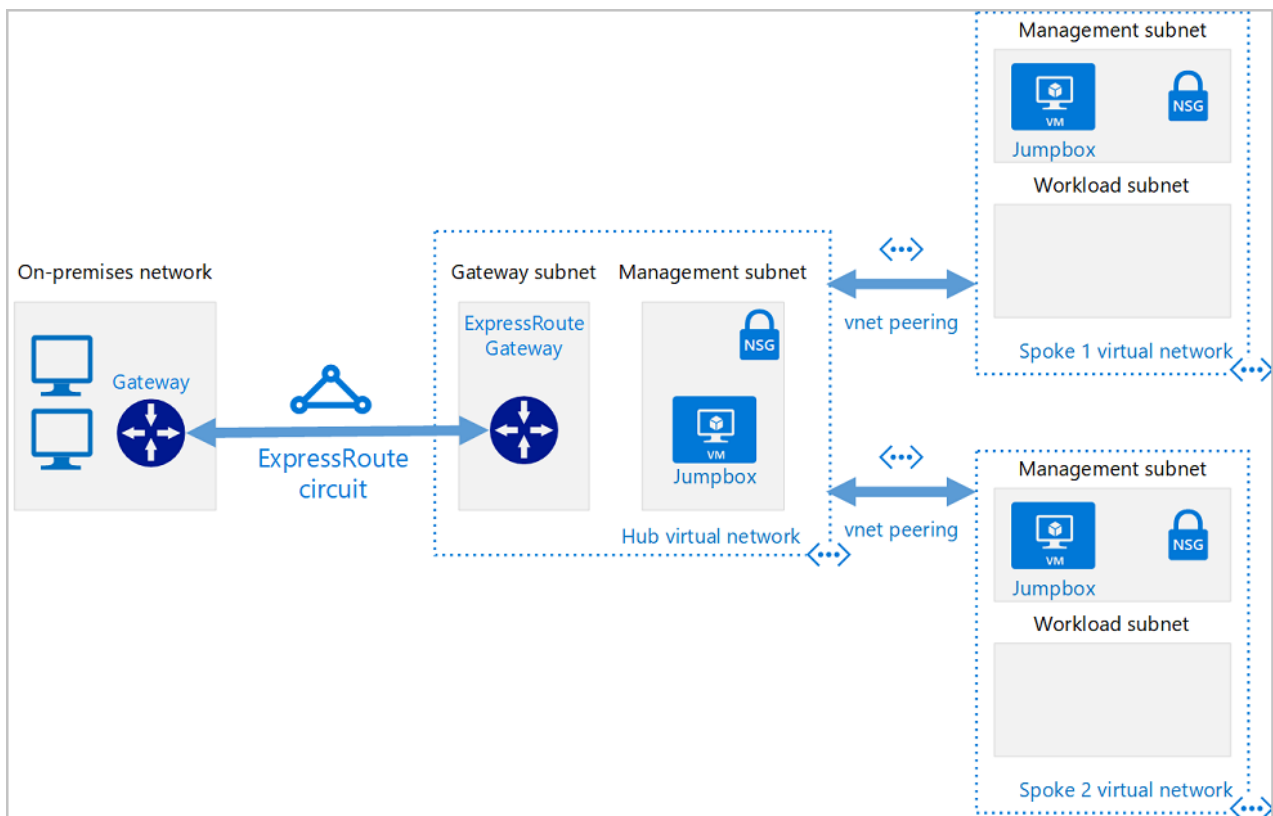
- Use HCL (HashiCorp Language) to lay out hub and spoke hybrid network reference architecture resources
- Use Terraform to create hub network appliance resources
- Use Terraform to create hub network in Azure to act as common point for all resources
- Use Terraform to create individual workloads as spoke VNets in Azure
- Use Terraform to establish gateways and connections between on premises and Azure networks
- Use Terraform to create VNet peerings to spoke networks

## Prerequisites

- **Azure subscription:** If you don't already have an Azure subscription, create a [free Azure account](#) before you begin.
- **Install and configure Terraform:** To provision VMs and other infrastructure in Azure, [install and configure Terraform](#)

## Hub and spoke topology architecture

In the hub and spoke topology, the hub is a VNet. The VNet acts as a central point of connectivity to your on-premises network. The spokes are VNets that peer with the hub, and can be used to isolate workloads. Traffic flows between the on-premises datacenter and the hub through an ExpressRoute or VPN gateway connection. The following image demonstrates the components in a hub and spoke topology:



## Benefits of the hub and spoke topology

A hub and spoke network topology is a way to isolate workloads while sharing common services. These services include identity and security. The hub is a VNet that acts as a central connection point to an on-premises network. The spokes are VNets that peer with the hub. Shared services are deployed in the hub, while individual workloads are deployed inside spoke networks. Here are some benefits of the hub and spoke network topology:

- **Cost savings** by centralizing services in a single location that can be shared by multiple workloads. These workloads include network virtual appliances and DNS servers.
- **Overcome subscriptions limits** by peering VNets from different subscriptions to the central hub.
- **Separation of concerns** between central IT (SecOps, InfraOps) and workloads (DevOps).

## Typical uses for the hub and spoke architecture

Some of the typical uses for a hub and spoke architecture include:

- Many customers have workloads that are deployed in different environments. These environments include development, testing, and production. Many times, these workloads need to share services such as DNS, IDS, NTP, or AD DS. These shared services can be placed in the hub VNet. That way, each environment is deployed to a spoke to maintain isolation.
- Workloads that don't require connectivity to each other, but require access to shared services.
- Enterprises that require central control over security aspects.
- Enterprises that require segregated management for the workloads in each spoke.

## Preview the demo components

As you work through each tutorial in this series, various components are defined in distinct Terraform scripts. The demo architecture created and deployed consists of the following components:

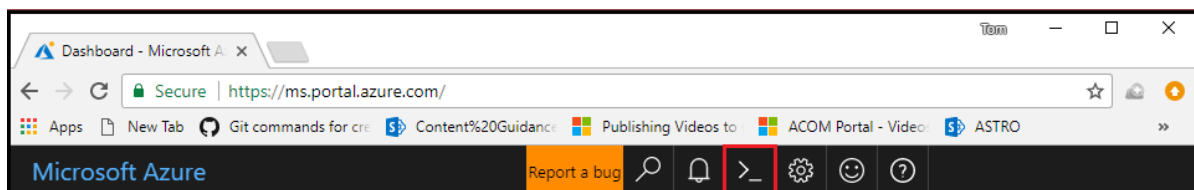
- **On-premises network.** A private local-area network running with an organization. For hub and spoke reference architecture, a VNet in Azure is used to simulate an on-premises network.

- **VPN device.** A VPN device or service provides external connectivity to the on-premises network. The VPN device may be a hardware appliance or a software solution.
- **Hub VNet.** The hub is the central point of connectivity to your on-premises network and a place to host services. These services can be consumed by the different workloads hosted in the spoke VNets.
- **Gateway subnet.** The VNet gateways are held in the same subnet.
- **Spoke VNets.** Spokes can be used to isolate workloads in their own VNets, managed separately from other spokes. Each workload might include multiple tiers, with multiple subnets connected through Azure load balancers.
- **VNet peering.** Two VNets can be connected using a peering connection. Peering connections are non-transitive, low latency connections between VNets. Once peered, the VNets exchange traffic by using the Azure backbone, without needing a router. In a hub and spoke network topology, VNet peering is used to connect the hub to each spoke. You can peer VNets in the same region, or different regions.

## Create the directory structure

Create the directory that holds your Terraform configuration files for the demo.

1. Browse to the [Azure portal](https://ms.portal.azure.com/).
2. Open [Azure Cloud Shell](#). If you didn't select an environment previously, select **Bash** as your environment.



3. Change directories to the `clouddrive` directory.

```
cd clouddrive
```

4. Create a directory named `hub-spoke`.

```
mkdir hub-spoke
```

5. Change directories to the new directory:

```
cd hub-spoke
```

## Declare the Azure provider

Create the Terraform configuration file that declares the Azure provider.

1. In Cloud Shell, open a new file named `main.tf`.

```
code main.tf
```

2. Paste the following code into the editor:

```
provider "azurerm" {  
  version = "~>1.22"  
}
```

3. Save the file and exit the editor.

## Create the variables file

Create the Terraform configuration file for common variables that are used across different scripts.

1. In Cloud Shell, open a new file named `variables.tf`.

```
code variables.tf
```

2. Paste the following code into the editor:

```
variable "location" {  
  description = "Location of the network"  
  default     = "centralus"  
}  
  
variable "username" {  
  description = "Username for Virtual Machines"  
  default     = "testadmin"  
}  
  
variable "password" {  
  description = "Password for Virtual Machines"  
  default     = "Password1234!"  
}  
  
variable "vm_size" {  
  description = "Size of the VMs"  
  default     = "Standard_DS1_v2"  
}
```

3. Save the file and exit the editor.

## Next steps

[Create on-premises virtual network with Terraform in Azure](#)

This tutorial shows how to implement an on-premises network using an Azure virtual network (VNet). An Azure VNet could be replaced by your own private virtual network. To do so, map the appropriate IP addresses in the subnets.

The following tasks are explained:

- Use HCL (HashiCorp Language) to implement an on-premises VNet in hub-spoke topology
- Use Terraform to create hub network appliance resources
- Use Terraform to create on-premises virtual machine
- Use Terraform to create on-premises virtual private network gateway

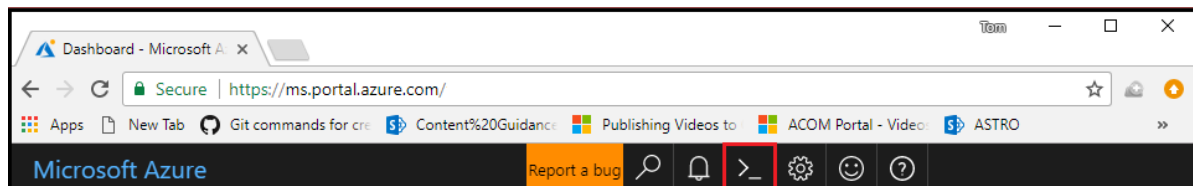
## Prerequisites

1. [Create a hub and spoke hybrid network topology with Terraform in Azure.](#)

## Create the directory structure

To simulate an on-premises network, create an Azure virtual network. The demo VNet takes the place of an actual private on-premises network. To do the same with your existing on-premises network, map appropriate IP addresses in the subnets.

1. Browse to the [Azure portal](#).
2. Open [Azure Cloud Shell](#). If you didn't select an environment previously, select **Bash** as your environment.



3. Change directories to the `clouddrive` directory.

```
cd clouddrive
```

4. Change directories to the new directory:

```
cd hub-spoke
```

## Declare the on-premises VNet

Create the Terraform configuration file that declares an on-premises VNet.

1. In Cloud Shell, open a new file named `on-prem.tf`.

```
code on-prem.tf
```

## 2. Paste the following code into the editor:

```
locals {
  onprem-location      = "SouthCentralUS"
  onprem-resource-group = "onprem-vnet-rg"
  prefix-onprem        = "onprem"
}

resource "azurerm_resource_group" "onprem-vnet-rg" {
  name     = local.onprem-resource-group
  location = local.onprem-location
}

resource "azurerm_virtual_network" "onprem-vnet" {
  name                = "onprem-vnet"
  location            = azurerm_resource_group.onprem-vnet-rg.location
  resource_group_name = azurerm_resource_group.onprem-vnet-rg.name
  address_space       = ["192.168.0.0/16"]

  tags {
    environment = local.prefix-onprem
  }
}

resource "azurerm_subnet" "onprem-gateway-subnet" {
  name                = "GatewaySubnet"
  resource_group_name = azurerm_resource_group.onprem-vnet-rg.name
  virtual_network_name = azurerm_virtual_network.onprem-vnet.name
  address_prefix       = "192.168.255.224/27"
}

resource "azurerm_subnet" "onprem-mgmt" {
  name                = "mgmt"
  resource_group_name = azurerm_resource_group.onprem-vnet-rg.name
  virtual_network_name = azurerm_virtual_network.onprem-vnet.name
  address_prefix       = "192.168.1.128/25"
}

resource "azurerm_public_ip" "onprem-pip" {
  name                = "${local.prefix-onprem}-pip"
  location            = azurerm_resource_group.onprem-vnet-rg.location
  resource_group_name = azurerm_resource_group.onprem-vnet-rg.name
  allocation_method   = "Dynamic"

  tags {
    environment = local.prefix-onprem
  }
}

resource "azurerm_network_interface" "onprem-nic" {
  name                = "${local.prefix-onprem}-nic"
  location            = azurerm_resource_group.onprem-vnet-rg.location
  resource_group_name = azurerm_resource_group.onprem-vnet-rg.name
  enable_ip_forwarding = true

  ip_configuration {
    name                = local.prefix-onprem
    subnet_id           = azurerm_subnet.onprem-mgmt.id
    private_ip_address_allocation = "Dynamic"
    public_ip_address_id = azurerm_public_ip.onprem-pip.id
  }
}

# Create Network Security Group and rule
resource "azurerm_network_security_group" "onprem-nsg" {
  name                = "${local.prefix-onprem}-nsg"
  location            = azurerm_resource_group.onprem-vnet-rg.location
  resource_group_name = azurerm_resource_group.onprem-vnet-rg.name
```



```

    security_rule {
      name           = "SSH"
      priority       = 1001
      direction      = "Inbound"
      access         = "Allow"
      protocol       = "Tcp"
      source_port_range = "*"
      destination_port_range = "22"
      source_address_prefix = "*"
      destination_address_prefix = "*"
    }

    tags {
      environment = "onprem"
    }
  }

  resource "azurerm_subnet_network_security_group_association" "mgmt-nsg-association" {
    subnet_id           = azurerm_subnet.onprem-mgmt.id
    network_security_group_id = azurerm_network_security_group.onprem-nsg.id
  }

  resource "azurerm_virtual_machine" "onprem-vm" {
    name                = "${local.prefix-onprem}-vm"
    location            = azurerm_resource_group.onprem-vnet-rg.location
    resource_group_name = azurerm_resource_group.onprem-vnet-rg.name
    network_interface_ids = [azurerm_network_interface.onprem-nic.id]
    vm_size            = var.vmsize

    storage_image_reference {
      publisher = "Canonical"
      offer     = "UbuntuServer"
      sku       = "16.04-LTS"
      version   = "latest"
    }

    storage_os_disk {
      name          = "myosdisk1"
      caching       = "ReadWrite"
      create_option = "FromImage"
      managed_disk_type = "Standard_LRS"
    }

    os_profile {
      computer_name = "${local.prefix-onprem}-vm"
      admin_username = var.username
      admin_password = var.password
    }

    os_profile_linux_config {
      disable_password_authentication = false
    }

    tags {
      environment = local.prefix-onprem
    }
  }

  resource "azurerm_public_ip" "onprem-vpn-gateway1-pip" {
    name                = "${local.prefix-onprem}-vpn-gateway1-pip"
    location            = azurerm_resource_group.onprem-vnet-rg.location
    resource_group_name = azurerm_resource_group.onprem-vnet-rg.name

    allocation_method = "Dynamic"
  }

  resource "azurerm_virtual_network_gateway" "onprem-vpn-gateway" {
    name                = "onprem-vpn-gateway1"

```

```

name          = onprem-vpn-gateway1
location      = azurerm_resource_group.onprem-vnet-rg.location
resource_group_name = azurerm_resource_group.onprem-vnet-rg.name

type          = "Vpn"
vpn_type      = "RouteBased"

active_active = false
enable_bgp    = false
sku           = "VpnGw1"

ip_configuration {
  name                = "vnetGatewayConfig"
  public_ip_address_id = azurerm_public_ip.onprem-vpn-gateway1-pip.id
  private_ip_address_allocation = "Dynamic"
  subnet_id           = azurerm_subnet.onprem-gateway-subnet.id
}
depends_on = ["azurerm_public_ip.onprem-vpn-gateway1-pip"]
}

```

3. Save the file and exit the editor.

## Next steps

[Create a hub virtual network with Terraform in Azure](#)

The hub virtual network acts as the central point of connectivity to the on-premises network. The virtual network hosts shared services consumed by workloads hosted in the spoke virtual networks. For demo purposes, no shared services are implemented in this tutorial.

This tutorial covers the following tasks:

- Use the HashiCorp Configuration Language (HCL) to implement the hub virtual network in a hub-and-spoke topology.
- Use Terraform to create a hub jumpbox virtual machine.
- Use Terraform to create a hub virtual private network gateway.
- Use Terraform to create hub and on-premises gateway connections.

## Prerequisites

1. [Create a hub-and-spoke hybrid network topology with Terraform in Azure.](#)
2. [Create an on-premises virtual network with Terraform in Azure.](#)

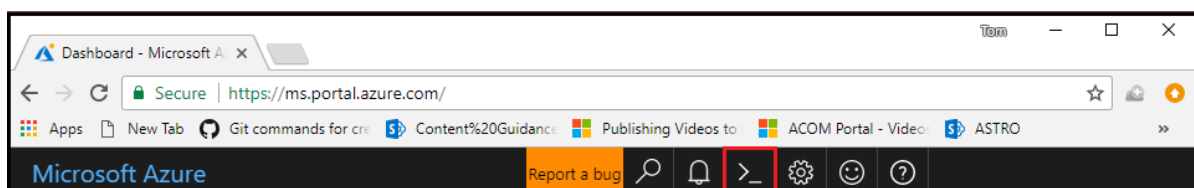
## Create the directory structure

The hub network consists of the following components:

- A hub virtual network
- A hub virtual network gateway
- Hub gateway connections

The following Terraform configuration file defines the resources:

1. Browse to the [Azure portal](#).
2. Open [Azure Cloud Shell](#). If you didn't select an environment previously, select **Bash** as your environment.



3. Change directories to the `clouddrive` directory.

```
cd clouddrive
```

4. Change directories to the new directory.

```
cd hub-spoke
```

## Declare the hub virtual network

Create the Terraform configuration file that declares the hub virtual network.

1. In Cloud Shell, create a file named `hub-vnet.tf`.

```
code hub-vnet.tf
```

2. Paste the following code into the editor:

```
locals {
  prefix-hub      = "hub"
  hub-location    = "CentralUS"
  hub-resource-group = "hub-vnet-rg"
  shared-key      = "4-v3ry-53cr37-1p53c-5h4r3d-k3y"
}

resource "azurerm_resource_group" "hub-vnet-rg" {
  name     = local.hub-resource-group
  location = local.hub-location
}

resource "azurerm_virtual_network" "hub-vnet" {
  name                = "${local.prefix-hub}-vnet"
  location            = azurerm_resource_group.hub-vnet-rg.location
  resource_group_name = azurerm_resource_group.hub-vnet-rg.name
  address_space       = ["10.0.0.0/16"]

  tags {
    environment = "hub-spoke"
  }
}

resource "azurerm_subnet" "hub-gateway-subnet" {
  name                 = "GatewaySubnet"
  resource_group_name = azurerm_resource_group.hub-vnet-rg.name
  virtual_network_name = azurerm_virtual_network.hub-vnet.name
  address_prefix       = "10.0.255.224/27"
}

resource "azurerm_subnet" "hub-mgmt" {
  name                 = "mgmt"
  resource_group_name = azurerm_resource_group.hub-vnet-rg.name
  virtual_network_name = azurerm_virtual_network.hub-vnet.name
  address_prefix       = "10.0.0.64/27"
}

resource "azurerm_subnet" "hub-dmz" {
  name                 = "dmz"
  resource_group_name = azurerm_resource_group.hub-vnet-rg.name
  virtual_network_name = azurerm_virtual_network.hub-vnet.name
  address_prefix       = "10.0.0.32/27"
}

resource "azurerm_network_interface" "hub-nic" {
  name                = "${local.prefix-hub}-nic"
  location            = azurerm_resource_group.hub-vnet-rg.location
  resource_group_name = azurerm_resource_group.hub-vnet-rg.name
  enable_ip_forwarding = true

  ip_configuration {
    name                 = local.prefix-hub
    subnet_id            = azurerm_subnet.hub-mgmt.id
    private_ip_address_allocation = "Dynamic"
  }

  tags {
    environment = local.prefix-hub
  }
}
```

```

#Virtual Machine
resource "azurerm_virtual_machine" "hub-vm" {
  name                = "${local.prefix-hub}-vm"
  location            = azurerm_resource_group.hub-vnet-rg.location
  resource_group_name = azurerm_resource_group.hub-vnet-rg.name
  network_interface_ids = [azurerm_network_interface.hub-nic.id]
  vm_size             = var.vmsize

  storage_image_reference {
    publisher = "Canonical"
    offer     = "UbuntuServer"
    sku       = "16.04-LTS"
    version   = "latest"
  }

  storage_os_disk {
    name          = "myosdisk1"
    caching       = "ReadWrite"
    create_option = "FromImage"
    managed_disk_type = "Standard_LRS"
  }

  os_profile {
    computer_name = "${local.prefix-hub}-vm"
    admin_username = var.username
    admin_password = var.password
  }

  os_profile_linux_config {
    disable_password_authentication = false
  }

  tags {
    environment = local.prefix-hub
  }
}

# Virtual Network Gateway
resource "azurerm_public_ip" "hub-vpn-gateway1-pip" {
  name                = "hub-vpn-gateway1-pip"
  location            = azurerm_resource_group.hub-vnet-rg.location
  resource_group_name = azurerm_resource_group.hub-vnet-rg.name

  allocation_method = "Dynamic"
}

resource "azurerm_virtual_network_gateway" "hub-vnet-gateway" {
  name                = "hub-vpn-gateway1"
  location            = azurerm_resource_group.hub-vnet-rg.location
  resource_group_name = azurerm_resource_group.hub-vnet-rg.name

  type      = "Vpn"
  vpn_type  = "RouteBased"

  active_active = false
  enable_bgp    = false
  sku           = "VpnGw1"

  ip_configuration {
    name                = "vnetGatewayConfig"
    public_ip_address_id = azurerm_public_ip.hub-vpn-gateway1-pip.id
    private_ip_address_allocation = "Dynamic"
    subnet_id           = azurerm_subnet.hub-gateway-subnet.id
  }
  depends_on = ["azurerm_public_ip.hub-vpn-gateway1-pip"]
}

resource "azurerm_virtual_network_gateway_connection" "hub-onprem-conn" {

```

```

    name                = "hub-onprem-conn"
    location             = azurerm_resource_group.hub-vnet-rg.location
    resource_group_name = azurerm_resource_group.hub-vnet-rg.name

    type                = "Vnet2Vnet"
    routing_weight = 1

    virtual_network_gateway_id      = azurerm_virtual_network_gateway.hub-vnet-gateway.id
    peer_virtual_network_gateway_id = azurerm_virtual_network_gateway.onprem-vpn-gateway.id

    shared_key = local.shared-key
}

resource "azurerm_virtual_network_gateway_connection" "onprem-hub-conn" {
    name                = "onprem-hub-conn"
    location            = azurerm_resource_group.onprem-vnet-rg.location
    resource_group_name = azurerm_resource_group.onprem-vnet-rg.name
    type                = "Vnet2Vnet"
    routing_weight = 1
    virtual_network_gateway_id      = azurerm_virtual_network_gateway.onprem-vpn-gateway.id
    peer_virtual_network_gateway_id = azurerm_virtual_network_gateway.hub-vnet-gateway.id

    shared_key = local.shared-key
}

```

3. Save the file, and exit the editor.

## Next steps

[Create a hub virtual network appliance with Terraform in Azure](#)

A **VPN device** is a device that provides external connectivity to an on-premises network. The VPN device may be a hardware device or a software solution. One example of a software solution is Routing and Remote Access Service (RRAS) in Windows Server 2012. For more information about VPN appliances, see [About VPN devices for Site-to-Site VPN Gateway connections](#).

Azure supports a broad variety of network virtual appliances from which to select. For this tutorial, an Ubuntu image is used. To learn more about the broad variety of device solutions supported in Azure, see the [Network Appliances home page](#).

This tutorial covers the following tasks:

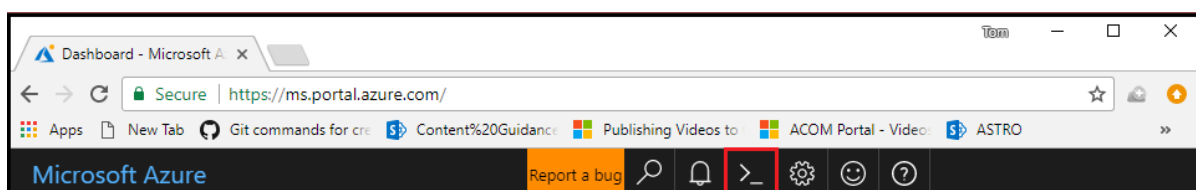
- Use HCL (HashiCorp Language) to implement the Hub VNet in hub-spoke topology
- Use Terraform to create Hub Network Virtual Machine which acts as appliance
- Use Terraform to enable routes using CustomScript extensions
- Use Terraform to create Hub and Spoke gateway route tables

## Prerequisites

1. [Create a hub and spoke hybrid network topology with Terraform in Azure](#).
2. [Create on-premises virtual network with Terraform in Azure](#).
3. [Create a hub virtual network with Terraform in Azure](#).

## Create the directory structure

1. Browse to the [Azure portal](#).
2. Open [Azure Cloud Shell](#). If you didn't select an environment previously, select **Bash** as your environment.



3. Change directories to the `clooudrive` directory.

```
cd clooudrive
```

4. Change directories to the new directory:

```
cd hub-spoke
```

## Declare the hub network appliance

Create the Terraform configuration file that declares an on-premises virtual network.

1. In Cloud Shell, create a new file named `hub-nva.tf`.

```
code hub-nva.tf
```

2. Paste the following code into the editor:

```
locals {
  prefix-hub-nva      = "hub-nva"
  hub-nva-location    = "CentralUS"
  hub-nva-resource-group = "hub-nva-rg"
}

resource "azurerm_resource_group" "hub-nva-rg" {
  name     = "${local.prefix-hub-nva}-rg"
  location = local.hub-nva-location

  tags {
    environment = local.prefix-hub-nva
  }
}

resource "azurerm_network_interface" "hub-nva-nic" {
  name                 = "${local.prefix-hub-nva}-nic"
  location             = azurerm_resource_group.hub-nva-rg.location
  resource_group_name = azurerm_resource_group.hub-nva-rg.name
  enable_ip_forwarding = true

  ip_configuration {
    name                 = local.prefix-hub-nva
    subnet_id           = azurerm_subnet.hub-dmz.id
    private_ip_address_allocation = "Static"
    private_ip_address      = "10.0.0.36"
  }

  tags {
    environment = local.prefix-hub-nva
  }
}

resource "azurerm_virtual_machine" "hub-nva-vm" {
  name                 = "${local.prefix-hub-nva}-vm"
  location             = azurerm_resource_group.hub-nva-rg.location
  resource_group_name = azurerm_resource_group.hub-nva-rg.name
  network_interface_ids = [azurerm_network_interface.hub-nva-nic.id]
  vm_size              = var.vmsize

  storage_image_reference {
    publisher = "Canonical"
    offer     = "UbuntuServer"
    sku       = "16.04-LTS"
    version   = "latest"
  }

  storage_os_disk {
    name         = "myosdisk1"
    caching      = "ReadWrite"
    create_option = "FromImage"
    managed_disk_type = "Standard_LRS"
  }

  os_profile {
    computer_name = "${local.prefix-hub-nva}-vm"
    admin_username = var.username
    admin_password = var.password
  }

  os_profile_linux_config {
    disable_password_authentication = false
  }
}
```



```

    }

    tags {
        environment = local.prefix-hub-nva
    }
}

resource "azurerm_virtual_machine_extension" "enable-routes" {
    name                = "enable-iptables-routes"
    location             = azurerm_resource_group.hub-nva-rg.location
    resource_group_name = azurerm_resource_group.hub-nva-rg.name
    virtual_machine_name = azurerm_virtual_machine.hub-nva-vm.name
    publisher            = "Microsoft.Azure.Extensions"
    type                 = "CustomScript"
    type_handler_version = "2.0"

    settings = <<SETTINGS
    {
        "fileUri": [
            "https://raw.githubusercontent.com/mspnp/reference-architectures/master/scripts/linux/enable-
ip-forwarding.sh"
        ],
        "commandToExecute": "bash enable-ip-forwarding.sh"
    }
}
SETTINGS

    tags {
        environment = local.prefix-hub-nva
    }
}

resource "azurerm_route_table" "hub-gateway-rt" {
    name                = "hub-gateway-rt"
    location             = azurerm_resource_group.hub-nva-rg.location
    resource_group_name = azurerm_resource_group.hub-nva-rg.name
    disable_bgp_route_propagation = false

    route {
        name            = "toHub"
        address_prefix = "10.0.0.0/16"
        next_hop_type   = "VnetLocal"
    }

    route {
        name                = "toSpoke1"
        address_prefix      = "10.1.0.0/16"
        next_hop_type       = "VirtualAppliance"
        next_hop_in_ip_address = "10.0.0.36"
    }

    route {
        name                = "toSpoke2"
        address_prefix      = "10.2.0.0/16"
        next_hop_type       = "VirtualAppliance"
        next_hop_in_ip_address = "10.0.0.36"
    }

    tags {
        environment = local.prefix-hub-nva
    }
}

resource "azurerm_subnet_route_table_association" "hub-gateway-rt-hub-vnet-gateway-subnet" {
    subnet_id      = azurerm_subnet.hub-gateway-subnet.id
    route_table_id = azurerm_route_table.hub-gateway-rt.id
    depends_on    = ["azurerm_subnet.hub-gateway-subnet"]
}

resource "azurerm_route_table" "spoke1-rt" {

```

```

name = "spoke1-rt"
location = azurerm_resource_group.hub-nva-rg.location
resource_group_name = azurerm_resource_group.hub-nva-rg.name
disable_bgp_route_propagation = false

route {
  name = "toSpoke2"
  address_prefix = "10.2.0.0/16"
  next_hop_type = "VirtualAppliance"
  next_hop_in_ip_address = "10.0.0.36"
}

route {
  name = "default"
  address_prefix = "0.0.0.0/0"
  next_hop_type = "vnetlocal"
}

tags {
  environment = local.prefix-hub-nva
}
}

resource "azurerm_subnet_route_table_association" "spoke1-rt-spoke1-vnet-mgmt" {
  subnet_id = azurerm_subnet.spoke1-mgmt.id
  route_table_id = azurerm_route_table.spoke1-rt.id
  depends_on = ["azurerm_subnet.spoke1-mgmt"]
}

resource "azurerm_subnet_route_table_association" "spoke1-rt-spoke1-vnet-workload" {
  subnet_id = azurerm_subnet.spoke1-workload.id
  route_table_id = azurerm_route_table.spoke1-rt.id
  depends_on = ["azurerm_subnet.spoke1-workload"]
}

resource "azurerm_route_table" "spoke2-rt" {
  name = "spoke2-rt"
  location = azurerm_resource_group.hub-nva-rg.location
  resource_group_name = azurerm_resource_group.hub-nva-rg.name
  disable_bgp_route_propagation = false

  route {
    name = "toSpoke1"
    address_prefix = "10.1.0.0/16"
    next_hop_in_ip_address = "10.0.0.36"
    next_hop_type = "VirtualAppliance"
  }

  route {
    name = "default"
    address_prefix = "0.0.0.0/0"
    next_hop_type = "vnetlocal"
  }

  tags {
    environment = local.prefix-hub-nva
  }
}

resource "azurerm_subnet_route_table_association" "spoke2-rt-spoke2-vnet-mgmt" {
  subnet_id = azurerm_subnet.spoke2-mgmt.id
  route_table_id = azurerm_route_table.spoke2-rt.id
  depends_on = ["azurerm_subnet.spoke2-mgmt"]
}

resource "azurerm_subnet_route_table_association" "spoke2-rt-spoke2-vnet-workload" {
  subnet_id = azurerm_subnet.spoke2-workload.id
  route_table_id = azurerm_route_table.spoke2-rt.id
  depends_on = ["azurerm_subnet.spoke2-workload"]
}

```

```
    depends_on = [ azurerm_virtual_network ]
  }
```

3. Save the file and exit the editor.

## Next steps

[Create a spoke virtual networks with Terraform in Azure](#)

In this tutorial, you implement two separate spoke networks to demonstrate separation of workloads. The networks share common resources using hub virtual network. Spokes can be used to isolate workloads in their own VNets, managed separately from other spokes. Each workload might include multiple tiers, with multiple subnets connected through Azure load balancers.

This tutorial covers the following tasks:

- Use HCL (HashiCorp Language) to implement the Spoke VNets in hub-spoke topology
- Use Terraform to create Virtual machines in the spoke networks
- Use Terraform to establish virtual network peerings with the hub networks

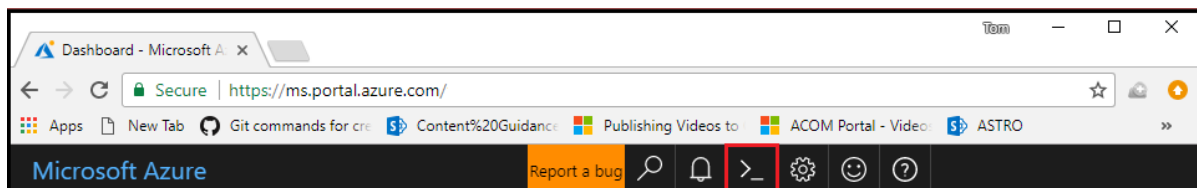
## Prerequisites

1. [Create a hub and spoke hybrid network topology with Terraform in Azure.](#)
2. [Create on-premises virtual network with Terraform in Azure.](#)
3. [Create a hub virtual network with Terraform in Azure.](#)
4. [Create a hub virtual network appliance with Terraform in Azure.](#)

## Create the directory structure

Two spoke scripts are created in this section. Each script defines a spoke virtual network and a virtual machine for the workload. A peered virtual network from hub to spoke is then created.

1. Browse to the [Azure portal](#).
2. Open [Azure Cloud Shell](#). If you didn't select an environment previously, select **Bash** as your environment.



3. Change directories to the `clouddrive` directory.

```
cd clouddrive
```

4. Change directories to the new directory:

```
cd hub-spoke
```

## Declare the two spoke networks

1. In Cloud Shell, open a new file named `spoke1.tf`.

```
code spoke1.tf
```

2. Paste the following code into the editor:

```
locals {
  spoke1-location      = "CentralUS"
  spoke1-resource-group = "spoke1-vnet-rg"
  prefix-spoke1        = "spoke1"
}

resource "azurerm_resource_group" "spoke1-vnet-rg" {
  name      = local.spoke1-resource-group
  location  = local.spoke1-location
}

resource "azurerm_virtual_network" "spoke1-vnet" {
  name                = "spoke1-vnet"
  location             = azurerm_resource_group.spoke1-vnet-rg.location
  resource_group_name = azurerm_resource_group.spoke1-vnet-rg.name
  address_space       = ["10.1.0.0/16"]

  tags {
    environment = local.prefix-spoke1
  }
}

resource "azurerm_subnet" "spoke1-mgmt" {
  name                = "mgmt"
  resource_group_name = azurerm_resource_group.spoke1-vnet-rg.name
  virtual_network_name = azurerm_virtual_network.spoke1-vnet.name
  address_prefix      = "10.1.0.64/27"
}

resource "azurerm_subnet" "spoke1-workload" {
  name                = "workload"
  resource_group_name = azurerm_resource_group.spoke1-vnet-rg.name
  virtual_network_name = azurerm_virtual_network.spoke1-vnet.name
  address_prefix      = "10.1.1.0/24"
}

resource "azurerm_virtual_network_peering" "spoke1-hub-peer" {
  name                = "spoke1-hub-peer"
  resource_group_name = azurerm_resource_group.spoke1-vnet-rg.name
  virtual_network_name = azurerm_virtual_network.spoke1-vnet.name
  remote_virtual_network_id = azurerm_virtual_network.hub-vnet.id

  allow_virtual_network_access = true
  allow_forwarded_traffic      = true
  allow_gateway_transit        = false
  use_remote_gateways          = true
  depends_on = ["azurerm_virtual_network.spoke1-vnet", "azurerm_virtual_network.hub-vnet",
"azurerm_virtual_network_gateway.hub-vnet-gateway"]
}

resource "azurerm_network_interface" "spoke1-nic" {
  name                = "${local.prefix-spoke1}-nic"
  location            = azurerm_resource_group.spoke1-vnet-rg.location
  resource_group_name = azurerm_resource_group.spoke1-vnet-rg.name
  enable_ip_forwarding = true

  ip_configuration {
    name                = local.prefix-spoke1
    subnet_id           = azurerm_subnet.spoke1-mgmt.id
    private_ip_address_allocation = "Dynamic"
  }
}

resource "azurerm_virtual_machine" "spoke1-vm" {
  name                = "${local.prefix-spoke1}-vm"
  location            = azurerm_resource_group.spoke1-vnet-rg.location
```

```

resource_group_name    = azurerm_resource_group.spoke1-vnet-rg.name
network_interface_ids = [azurerm_network_interface.spoke1-nic.id]
vm_size                = var.vmsize

storage_image_reference {
  publisher = "Canonical"
  offer     = "UbuntuServer"
  sku       = "16.04-LTS"
  version   = "latest"
}

storage_os_disk {
  name          = "myosdisk1"
  caching       = "ReadWrite"
  create_option = "FromImage"
  managed_disk_type = "Standard_LRS"
}

os_profile {
  computer_name = "${local.prefix-spoke1}-vm"
  admin_username = var.username
  admin_password = var.password
}

os_profile_linux_config {
  disable_password_authentication = false
}

tags {
  environment = local.prefix-spoke1
}
}

resource "azurerm_virtual_network_peering" "hub-spoke1-peer" {
  name                        = "hub-spoke1-peer"
  resource_group_name        = azurerm_resource_group.hub-vnet-rg.name
  virtual_network_name       = azurerm_virtual_network.hub-vnet.name
  remote_virtual_network_id = azurerm_virtual_network.spoke1-vnet.id
  allow_virtual_network_access = true
  allow_forwarded_traffic     = true
  allow_gateway_transit       = true
  use_remote_gateways         = false
  depends_on = ["azurerm_virtual_network.spoke1-vnet", "azurerm_virtual_network.hub-vnet",
"azurerm_virtual_network_gateway.hub-vnet-gateway"]
}

```

3. Save the file and exit the editor.

4. Create a new file named `spoke2.tf`.

```
code spoke2.tf
```

5. Paste the following code into the editor:

```

locals {
  spoke2-location      = "CentralUS"
  spoke2-resource-group = "spoke2-vnet-rg"
  prefix-spoke2        = "spoke2"
}

resource "azurerm_resource_group" "spoke2-vnet-rg" {
  name     = local.spoke2-resource-group
  location = local.spoke2-location
}

```

```

resource "azurerm_virtual_network" "spoke2-vnet" {
  name                = "${local.prefix-spoke2}-vnet"
  location            = azurerm_resource_group.spoke2-vnet-rg.location
  resource_group_name = azurerm_resource_group.spoke2-vnet-rg.name
  address_space       = ["10.2.0.0/16"]

  tags {
    environment = local.prefix-spoke2
  }
}

resource "azurerm_subnet" "spoke2-mgmt" {
  name                = "mgmt"
  resource_group_name = azurerm_resource_group.spoke2-vnet-rg.name
  virtual_network_name = azurerm_virtual_network.spoke2-vnet.name
  address_prefix       = "10.2.0.64/27"
}

resource "azurerm_subnet" "spoke2-workload" {
  name                = "workload"
  resource_group_name = azurerm_resource_group.spoke2-vnet-rg.name
  virtual_network_name = azurerm_virtual_network.spoke2-vnet.name
  address_prefix       = "10.2.1.0/24"
}

resource "azurerm_virtual_network_peering" "spoke2-hub-peer" {
  name                = "${local.prefix-spoke2}-hub-peer"
  resource_group_name = azurerm_resource_group.spoke2-vnet-rg.name
  virtual_network_name = azurerm_virtual_network.spoke2-vnet.name
  remote_virtual_network_id = azurerm_virtual_network.hub-vnet.id

  allow_virtual_network_access = true
  allow_forwarded_traffic      = true
  allow_gateway_transit        = false
  use_remote_gateways          = true
  depends_on = ["azurerm_virtual_network.spoke2-vnet", "azurerm_virtual_network.hub-vnet",
"azurerm_virtual_network_gateway.hub-vnet-gateway"]
}

resource "azurerm_network_interface" "spoke2-nic" {
  name                = "${local.prefix-spoke2}-nic"
  location            = azurerm_resource_group.spoke2-vnet-rg.location
  resource_group_name = azurerm_resource_group.spoke2-vnet-rg.name
  enable_ip_forwarding = true

  ip_configuration {
    name                = local.prefix-spoke2
    subnet_id           = azurerm_subnet.spoke2-mgmt.id
    private_ip_address_allocation = "Dynamic"
  }

  tags {
    environment = local.prefix-spoke2
  }
}

resource "azurerm_virtual_machine" "spoke2-vm" {
  name                = "${local.prefix-spoke2}-vm"
  location            = azurerm_resource_group.spoke2-vnet-rg.location
  resource_group_name = azurerm_resource_group.spoke2-vnet-rg.name
  network_interface_ids = [azurerm_network_interface.spoke2-nic.id]
  vm_size              = var.vmsize

  storage_image_reference {
    publisher = "Canonical"
    offer     = "UbuntuServer"
    sku       = "16.04-LTS"
    version   = "latest"
  }
}

```

```

storage_os_disk {
  name           = "myosdisk1"
  caching        = "ReadWrite"
  create_option  = "FromImage"
  managed_disk_type = "Standard_LRS"
}

os_profile {
  computer_name  = "${local.prefix-spoke2}-vm"
  admin_username = var.username
  admin_password = var.password
}

os_profile_linux_config {
  disable_password_authentication = false
}

tags {
  environment = local.prefix-spoke2
}
}

resource "azurerm_virtual_network_peering" "hub-spoke2-peer" {
  name                        = "hub-spoke2-peer"
  resource_group_name        = azurerm_resource_group.hub-vnet-rg.name
  virtual_network_name       = azurerm_virtual_network.hub-vnet.name
  remote_virtual_network_id  = azurerm_virtual_network.spoke2-vnet.id
  allow_virtual_network_access = true
  allow_forwarded_traffic    = true
  allow_gateway_transit      = true
  use_remote_gateways        = false
  depends_on = ["azurerm_virtual_network.spoke2-vnet", "azurerm_virtual_network.hub-vnet",
"azurerm_virtual_network_gateway.hub-vnet-gateway"]
}

```

6. Save the file and exit the editor.

## Next steps

[Validate a hub and spoke network with Terraform in Azure](#)



minutes to read • [Edit Online](#)

In this article, you execute the terraform files created in the previous article in this series. The result is a validation of the connectivity between the demo virtual networks.

This tutorial covers the following tasks:

- Use HCL (HashiCorp Language) to implement the Hub VNet in hub-spoke topology
- Use Terraform plan to verify the resources to be deployed
- Use Terraform apply to create the resources in Azure
- Verify the connectivity between different networks
- Use Terraform to destroy all the resources

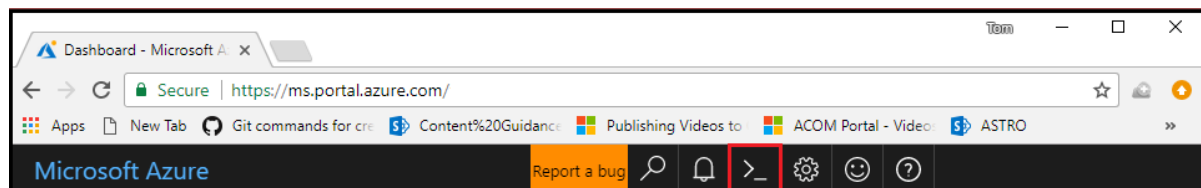
## Prerequisites

1. [Create a hub and spoke hybrid network topology with Terraform in Azure.](#)
2. [Create on-premises virtual network with Terraform in Azure.](#)
3. [Create a hub virtual network with Terraform in Azure.](#)
4. [Create a hub virtual network appliance with Terraform in Azure.](#)
5. [Create a spoke virtual networks with Terraform in Azure.](#)

## Verify your configuration

After completing the [prerequisites](#), verify the appropriate config files are present.

1. Browse to the [Azure portal](#).
2. Open [Azure Cloud Shell](#). If you didn't select an environment previously, select **Bash** as your environment.



3. Change directories to the `clouddrive` directory.

```
cd clouddrive
```

4. Change directories to the new directory:

```
cd hub-spoke
```

5. Run the `ls` command to verify that the `.tf` config files created in the previous tutorials are listed:

```
$ ls -al
total 122
drwxrwxrwx 2 root root    0 Mar  5 00:38
drwxrwxrwx 2 root root    0 Oct 25 15:41
-rwxrwxrwx 1 root root 6059 Mar  5 00:39 hub-nva.tf
-rwxrwxrwx 1 root root 5030 Mar  5 00:39 hub-vnet.tf
-rwxrwxrwx 1 root root  44 Mar  5 00:40 main.tf
-rwxrwxrwx 1 root root 5117 Mar  5 00:40 on-prem.tf
-rwxrwxrwx 1 root root 3845 Mar  5 00:40 spoke1.tf
-rwxrwxrwx 1 root root 3933 Mar  5 00:41 spoke2.tf
drwxrwxrwx 2 root root    0 Mar  5 00:42 terraform
-rwxrwxrwx 1 root root 99264 Mar  5 01:06 terraform.tfstate
-rwxrwxrwx 1 root root  471 Mar  5 00:41 variables.tf
```

## Deploy the resources

1. Initialize the Terraform provider:

```
terraform init
```

```
Initializing provider plugins...
- Checking for available provider plugins on https://releases.hashicorp.com...
- Downloading plugin for provider "azurerm" (1.22.1)...

Terraform has been successfully initialized!

You may now begin working with Terraform. Try running "terraform plan" to see
any changes that are required for your infrastructure. All Terraform commands
should now work.

If you ever set or change modules or backend configuration for Terraform,
rerun this command to reinitialize your working directory. If you forget, other
commands will detect it and remind you to do so if necessary.
```

2. Run the `terraform plan` command to see the effect of the deployment before execution:

```
terraform plan
```

```
allow_virtual_network_access: true
name: "spoke2-hub-peer"
remote_virtual_network_id: "${azurerm_virtual_network.hub-vnet.id}"
resource_group_name: "spoke2-vnet-rg"
use_remote_gateways: "true"
virtual_network_name: "spoke2-vnet"

Plan: 50 to add, 0 to change, 0 to destroy.

-----

Note: You didn't specify an "-out" parameter to save this plan, so Terraform
can't guarantee that exactly these actions will be performed if
"terraform apply" is subsequently run.
```

3. Deploy the solution:

```
terraform apply
```

Enter `yes` when prompted to confirm the deployment.

```
allow_virtual_network_access: true
name: "spoke2-hub-peer"
remote_virtual_network_id: "${azurerm_virtual_network.hub-vnet.id}"
resource_group_name: "spoke2-vnet-rg"
use_remote_gateways: "true"
virtual_network_name: "spoke2-vnet"

Plan: 50 to add, 0 to change, 0 to destroy.

Do you want to perform these actions?
Terraform will perform the actions described above.
Only 'yes' will be accepted to approve.

Enter a value: yes
```

## Test the hub VNet and spoke VNets

This section shows how to test connectivity from the simulated on-premises environment to the hub VNet.

1. In the Azure portal, browse to the **onprem-vnet-rg** resource group.
2. In the **onprem-vnet-rg** tab, select the VM named **onprem-vm**.
3. Select **Connect**.
4. Next to the text **Login using VM local account**, copy the **ssh** command to the clipboard.
5. From a Linux prompt, run `ssh` to connect to the simulated on-premises environment. Use the password specified in the `on-prem.tf` parameter file.
6. Run the `ping` command to test connectivity to the jumpbox VM in the hub VNet:

```
ping 10.0.0.68
```

7. Run the `ping` command to test connectivity to the jumpbox VMs in each spoke:

```
ping 10.1.0.68
ping 10.2.0.68
```

8. To exit the ssh session on the **onprem-vm** virtual machine, enter `exit` and press <Enter>.

## Troubleshoot VPN issues

For information about resolving VPN errors, see the article, [Troubleshoot a hybrid VPN connection](#).

## Clean up resources

When no longer needed, delete the resources created in the tutorial series.

1. Remove the resources declared in the plan:

```
terraform destroy
```

Enter `yes` when prompted to confirm the removal of the resources.

2. Change directories to the parent directory:

```
cd ..
```

3. Delete the `hub-scope` directory (including all of its files):

```
rm -r hub-spoke
```

## Next steps

[Learn more about using Terraform in Azure](#)

The Azure Terraform Visual Studio Code extension enables you to work with Terraform from the editor. With this extension, you can author, test, and run Terraform configurations. The extension also supports resource graph visualization.

In this article, you learn how to:

- Automate the provisioning of Azure services using Terraform
- install and use the Terraform Visual Studio Code extension for Azure services.
- Use Visual Studio Code to write, plan, and execute Terraform plans.

## Prerequisites

- **Azure subscription:** If you don't have an Azure subscription, create a [free account](#) before you begin.
- **Terraform:** [Install and configure Terraform](#).
- **Visual Studio Code:** Install the version of [Visual Studio Code](#) that is appropriate for your environment.

## Prepare your dev environment

### Install Git

To complete the exercises in the article, you need to [install Git](#).

### Install HashiCorp Terraform

Follow the instructions on the HashiCorp [Install Terraform](#) webpage, which covers:

- Downloading Terraform
- Installing Terraform
- Verifying Terraform is correctly installed

#### TIP

Be sure to follow the instructions regarding setting your PATH system variable.

### Install Node.js

To use Terraform in the Cloud Shell, you need to [install Node.js](#) 6.0+.

#### NOTE

To verify if Node.js is installed, open a terminal window and enter `node -v`.

### Install GraphViz

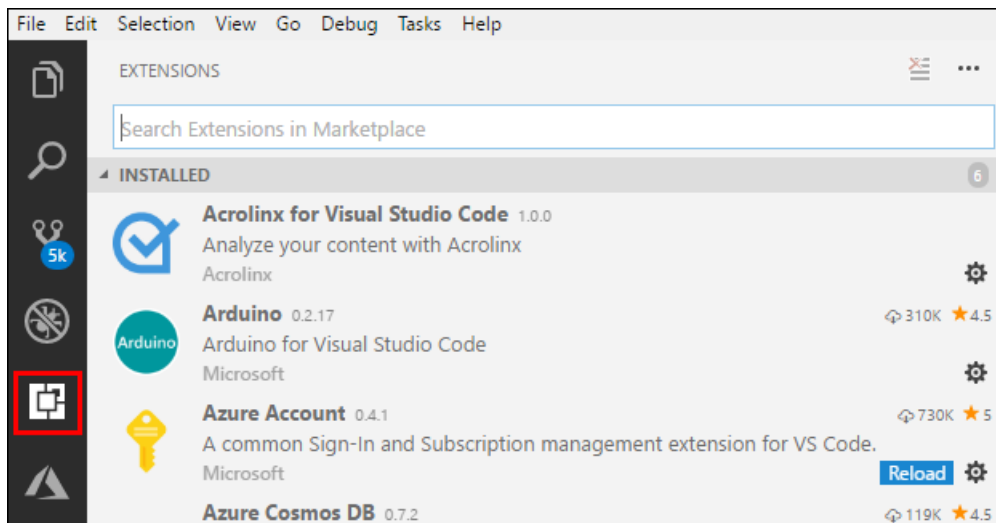
To use the Terraform visualize function, you need to [install GraphViz](#).

#### NOTE

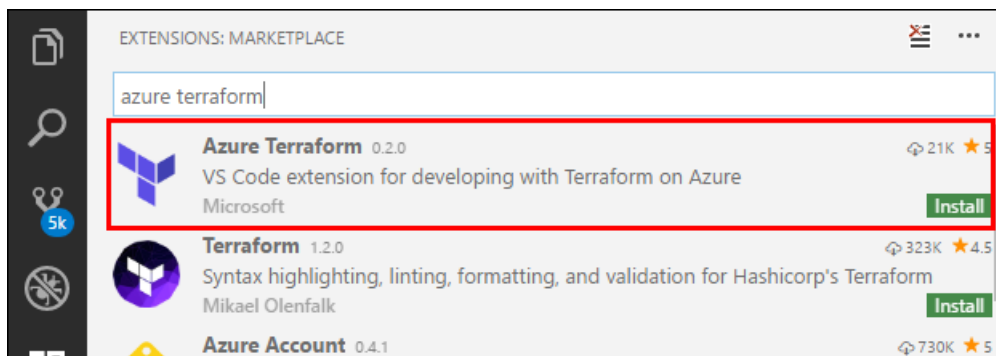
To verify if GraphViz is installed, open a terminal window and enter `dot -v`.

## Install the Azure Terraform Visual Studio Code extension

1. Launch Visual Studio Code.
2. Select Extensions.



3. Use the Search Extensions in Marketplace text box to search for the Azure Terraform extension:



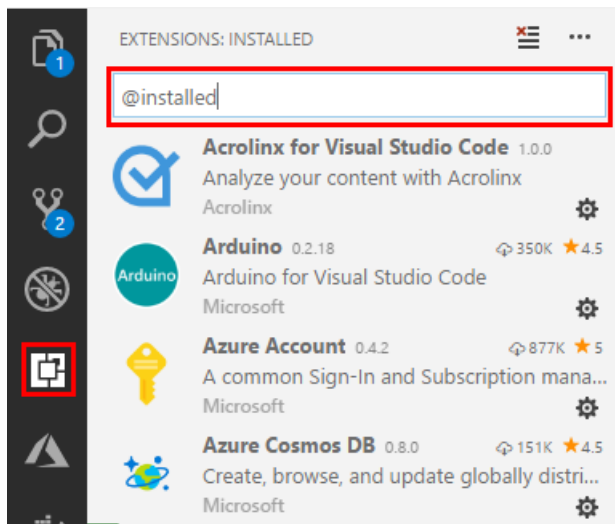
4. Select Install.

### NOTE

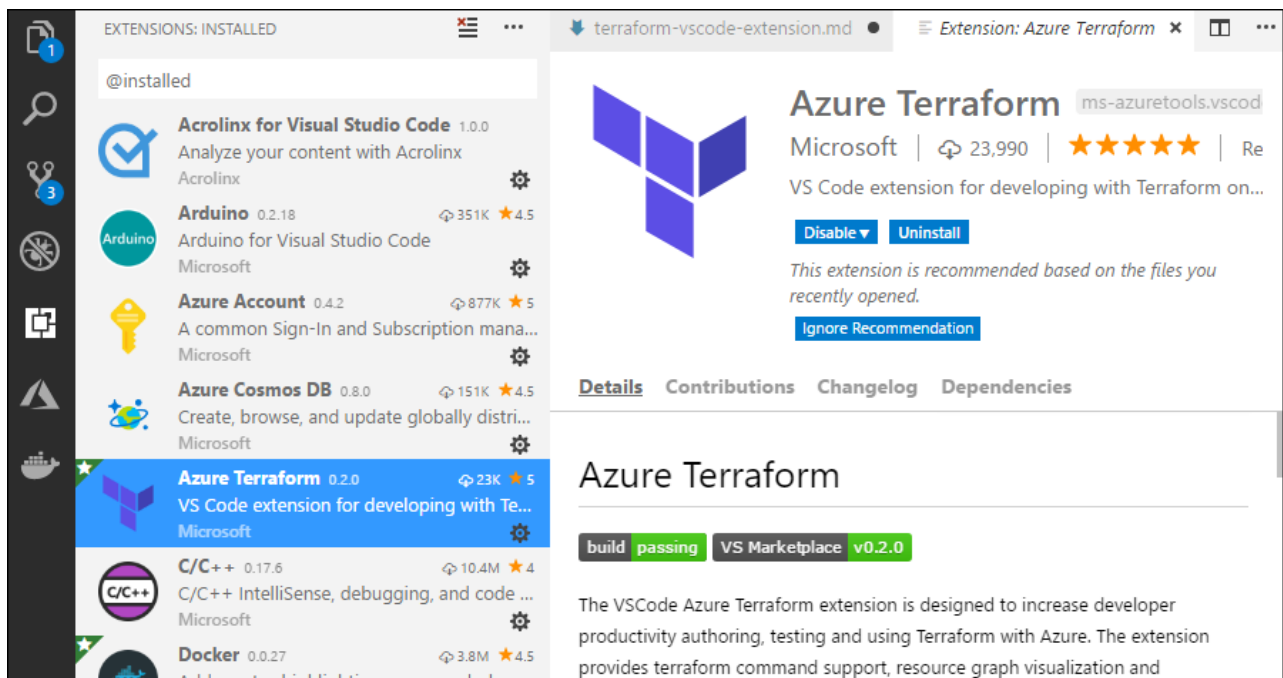
When you select **Install** for the Azure Terraform extension, Visual Studio Code will automatically install the Azure Account extension. Azure Account is a dependency file for the Azure Terraform extension, which it uses to perform Azure subscription authentications and Azure-related code extensions.

## Verify the Terraform extension is installed in Visual Studio Code

1. Select Extensions.
2. Enter `@installed` in the search text box.



The Azure Terraform extension will appear in the list of installed extensions.



You can now run all supported Terraform commands in your Cloud Shell environment from within Visual Studio Code.

## Exercise 1: Basic Terraform commands walk-through

In this exercise, you create and execute a basic Terraform configuration file that provisions a new Azure resource group.

### Prepare a test plan file

1. In Visual Studio Code, select **File** > **New File** from the menu bar.
2. In your browser, navigate to the [Terraform azurerm\\_resource\\_group page](#) and copy the code in the **Example Usage** code block:

Learn how Terraform fits into the

[Intro](#)
[Docs](#)
[Guides](#)
[Extend](#)
[Enterprise](#)
[Download](#)
[GitHub](#)

- > All Providers
- > Azure Provider
  - > Authenticating via the Azure CLI
  - > Authenticating via a Service Principal (Shared Account)
  - > Authenticating via Managed Service Identity
- > Data Sources
  - > azurerm\_application\_security\_group
  - > azurerm\_app\_service
  - > azurerm\_app\_service\_plan
  - > azurerm\_azuread\_application
  - > azurerm\_azuread\_service\_principal

## azurerm\_resource\_group

Manages a resource group on Azure.

### Example Usage

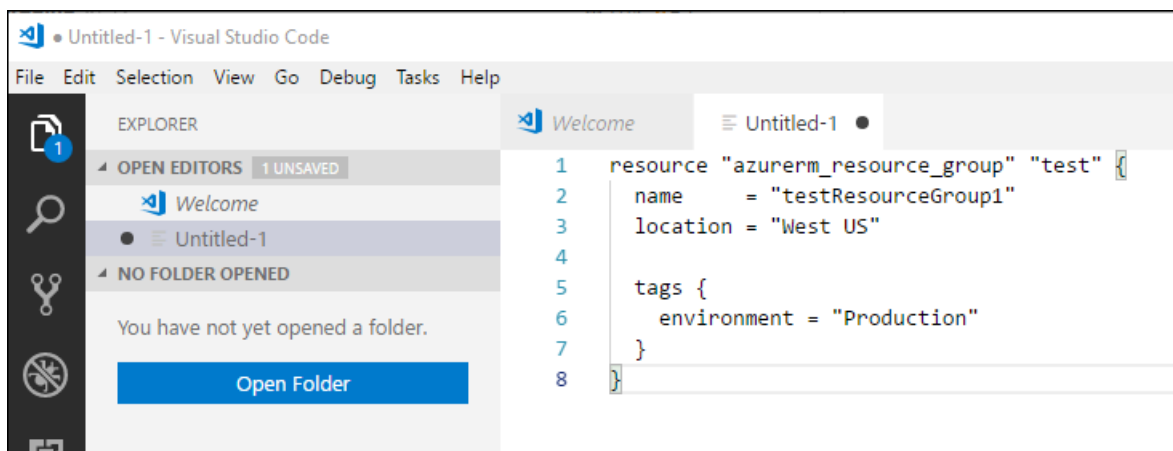
```

resource "azurerm_resource_group" "test" {
  name     = "testResourceGroup1"
  location = "West US"

  tags {
    environment = "Production"
  }
}

```

- Paste the copied code into the new file you created in Visual Studio Code.



#### NOTE

You may change the **name** value of the resource group, but it must be unique to your Azure subscription.

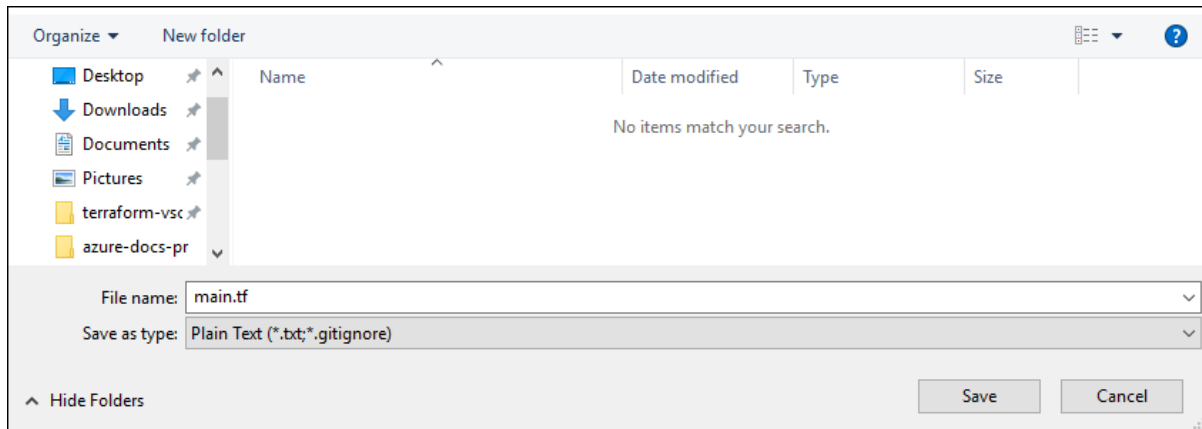
- From the menu bar, select **File > Save As**.
- In the **Save As** dialog, navigate to a location of your choice and then select **New folder**. (Change the name of the new folder to something more descriptive than *New folder*.)

#### NOTE

In this example, the folder is named **TERRAFORM-TEST-PLAN**.

- Make sure your new folder is highlighted (selected) and then select **Open**.
- In the **Save As** dialog, change the default name of the file to *main.tf*.





8. Select **Save**.

9. In the menu bar, select **File > Open Folder**. Navigate to and select the new folder you created.

### Run Terraform *init* command

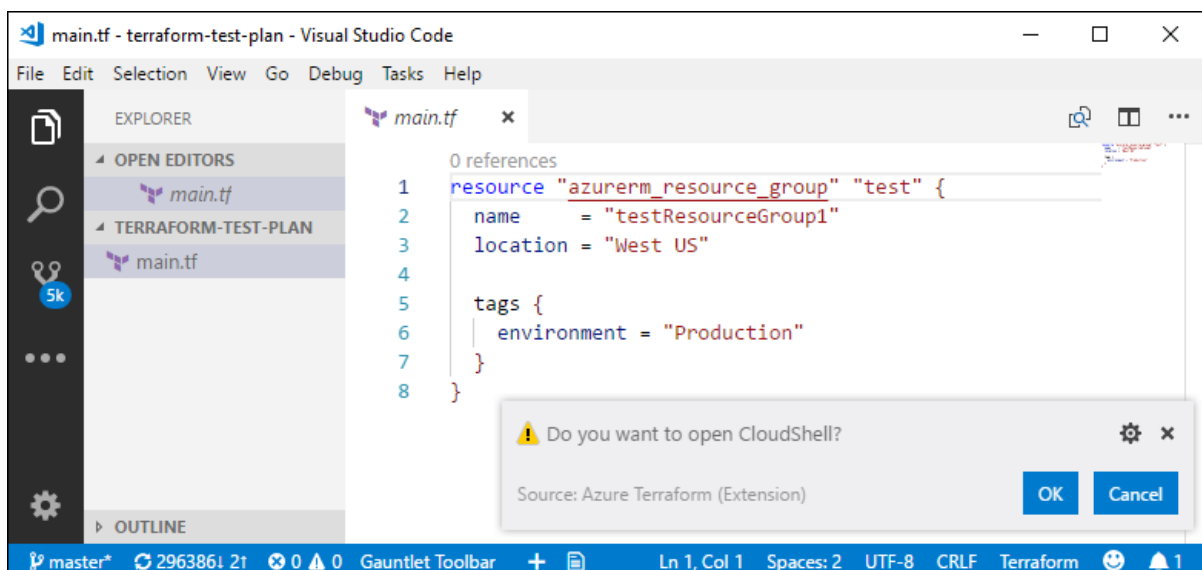
1. Launch Visual Studio Code.

2. From the Visual Studio Code menu bar, select **File > Open Folder...** and locate and select your *main.tf* file.

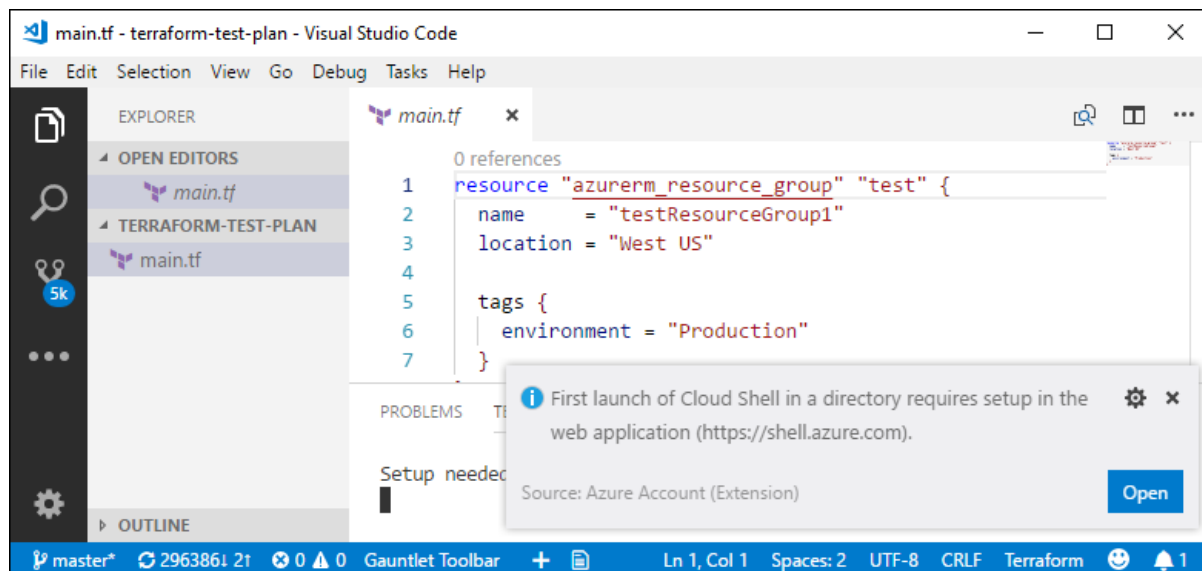


3. From the menu bar, select **View > Command Palette... > Azure Terraform: Init**.

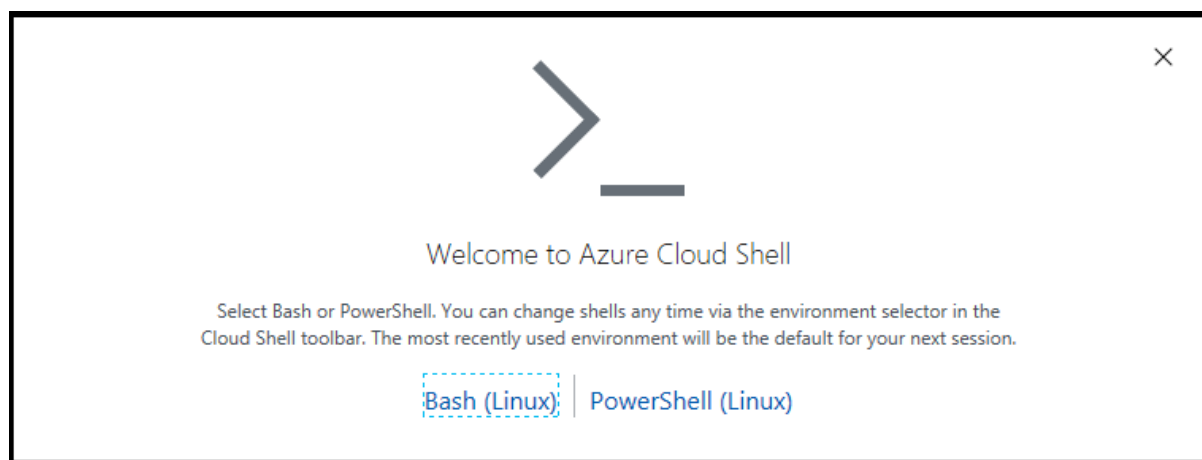
4. When the confirmation appears, select **OK**.



5. The first time you launch Cloud Shell from a new folder, you're prompted to create a web application. Select **Open**.



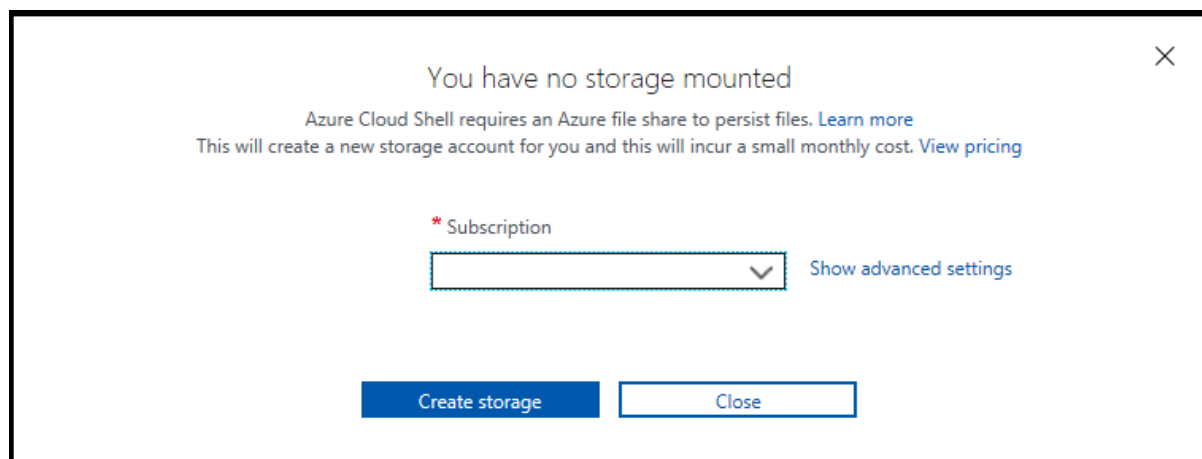
6. The Welcome to Azure Cloud Shell page opens. Select Bash or PowerShell.



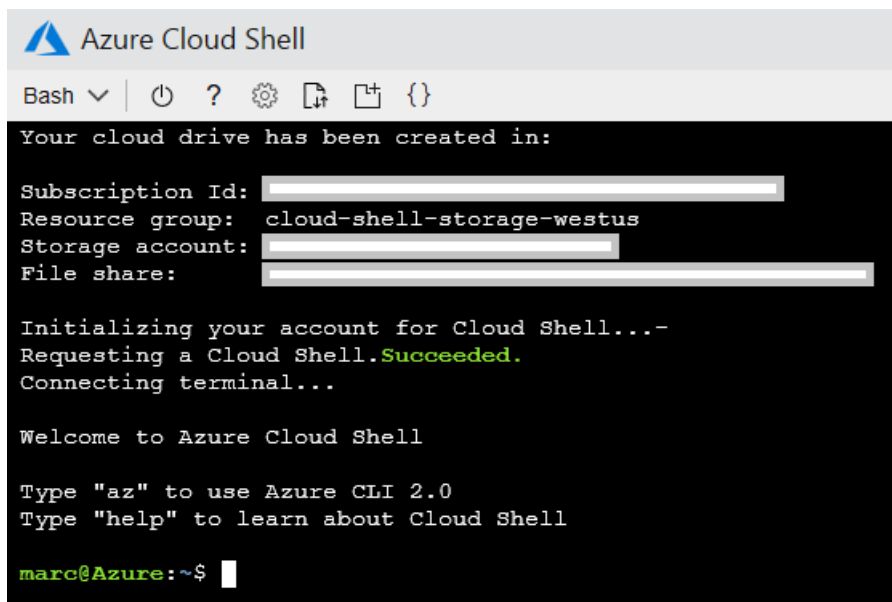
#### NOTE

In this example, Bash (Linux) was selected.

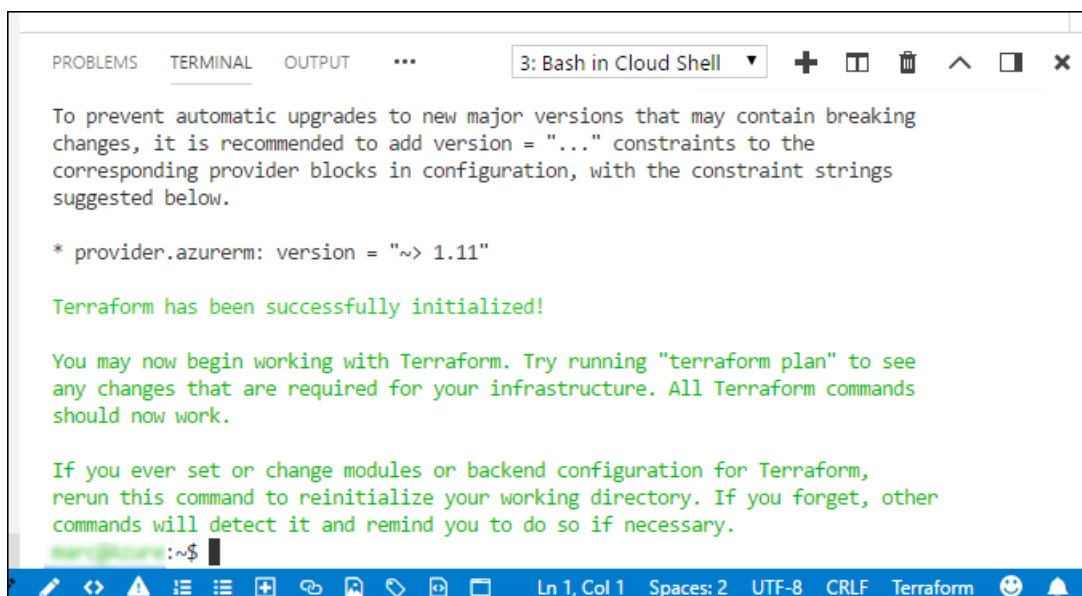
7. If you have not already set up an Azure storage account, the following screen appears. Select **Create storage**.



8. Azure Cloud Shell launches in the shell you previously selected and displays information for the cloud drive it just created for you.



9. You may now exit the Cloud Shell
10. From the menu bar, select **View > Command Palette > Azure Terraform: init.**



## Visualize the plan

Earlier in this tutorial, you installed GraphViz. Terraform can use GraphViz to generate a visual representation of either a configuration or execution plan. The Azure Terraform Visual Studio Code extension implements this feature via the *visualize* command.

- From the menu bar, select **View > Command Palette > Azure Terraform: Visualize.**



## Run Terraform *plan* command

The Terraform *plan* command is used to check whether the execution plan for a set of changes will do what you intended.

## NOTE

Terraform *plan* does not make any changes to your real Azure resources. To actually make the changes contained in your plan, we use the Terraform *apply* command.

- From the menu bar, select **View** > **Command Palette** > **Azure Terraform: plan**.

```
user@user:~$ cd "$HOME/cloudrive/terraform-test-plan"
bash: cd: /home/user/cloudrive/terraform-test-plan: No such file or directory
user@user:~$ terraform plan
Refreshing Terraform state in-memory prior to plan...
The refreshed state will be used to calculate this plan, but will not be
persisted to local or remote state storage.

-----

An execution plan has been generated and is shown below.
Resource actions are indicated with the following symbols:
  + create

Terraform will perform the following actions:

+ azurerm_resource_group.test
  id:          <computed>
  location:    "westus"
  name:        "testResourceGroup1"
  tags.%:      "1"
  tags.environment: "Production"

Plan: 1 to add, 0 to change, 0 to destroy.
```

## Run Terraform *apply* command

After being satisfied with the results of Terraform *plan*, you can run the *apply* command.

1. From the menu bar, select **View** > **Command Palette** > **Azure Terraform: apply**.

```
user@user:~$ terraform apply

An execution plan has been generated and is shown below.
Resource actions are indicated with the following symbols:
  + create

Terraform will perform the following actions:

+ azurerm_resource_group.test
  id:          <computed>
  location:    "westus"
  name:        "testResourceGroup1"
  tags.%:      "1"
  tags.environment: "Production"

Plan: 1 to add, 0 to change, 0 to destroy.

Do you want to perform these actions?
  Terraform will perform the actions described above.
  Only 'yes' will be accepted to approve.

Enter a value: █
```

2. Enter .

monopile: ~\$

1. From the menu bar, select **View > Command Palette > Azure Terraform: destroy**.

```

terraform destroy
bash: cd: /home/sarc/cloudDrive/terraform-test-plan: No such file or directory
sarc@arc:~$ terraform destroy
azure_rm_resource_group.test: Refreshing state... (ID: /subscriptions/8753d86c-a507-4867-b
889-.../resourceGroups/testResourceGroup1)

An execution plan has been generated and is shown below.
Resource actions are indicated with the following symbols:
  - destroy

Terraform will perform the following actions:

  - azure_rm_resource_group.test

Plan: 0 to add, 0 to change, 1 to destroy.

Do you really want to destroy?
  Terraform will destroy all your managed infrastructure, as shown above.
  There is no undo. Only 'yes' will be accepted to confirm.

Enter a value: █

```

2. Enter yes.

```

Do you really want to destroy?
  Terraform will destroy all your managed infrastructure, as shown above.
  There is no undo. Only 'yes' will be accepted to confirm.

Enter a value: yes

azure_rm_resource_group.test: Destroying... (ID: /subscriptions/8753d86c-a507-4867-b889-...
/resourceGroups/testResourceGroup1)
azure_rm_resource_group.test: Still destroying... (ID: /subscriptions/8753d86c-a507-4867-b
889-.../resourceGroups/testResourceGroup1, 10s elapsed)
azure_rm_resource_group.test: Still destroying... (ID: /subscriptions/8753d86c-a507-4867-b
889-.../resourceGroups/testResourceGroup1, 20s elapsed)
azure_rm_resource_group.test: Still destroying... (ID: /subscriptions/8753d86c-a507-4867-b
889-.../resourceGroups/testResourceGroup1, 30s elapsed)
azure_rm_resource_group.test: Still destroying... (ID: /subscriptions/8753d86c-a507-4867-b
889-.../resourceGroups/testResourceGroup1, 40s elapsed)
azure_rm_resource_group.test: Destruction complete after 46s

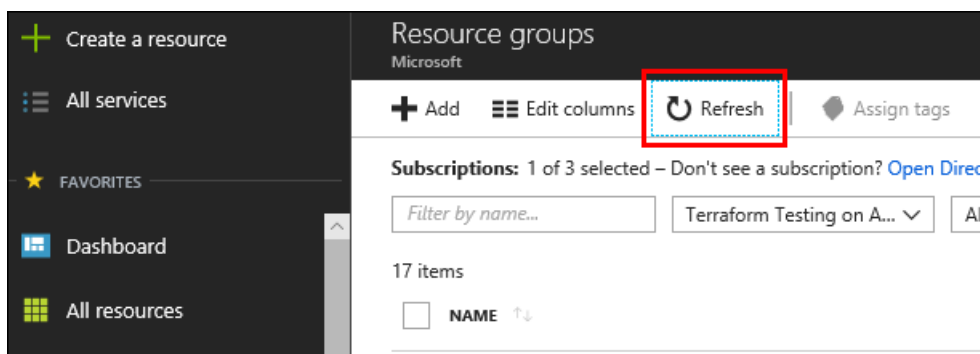
Destroy complete! Resources: 1 destroyed.
sarc@arc:~$ █

```

### Verify your resource group was destroyed

To confirm that Terraform successfully destroyed your new resource group:

1. Select **Refresh** on the Azure portal **Resource groups** page.
2. Your resource group will no longer be listed.

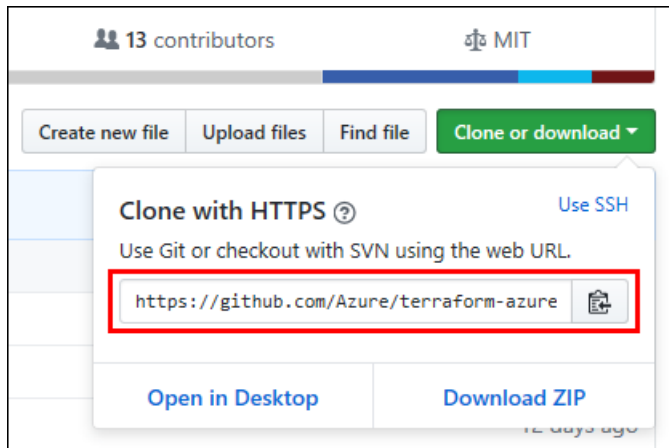


## Exercise 2: Terraform *compute* module

In this exercise, you learn how to load the Terraform *compute* module into the Visual Studio Code environment.

### Clone the terraform-azurerm-compute module

1. Use [this link](#) to access the Terraform Azure Rm Compute module on GitHub.
2. Select **Clone or download**.

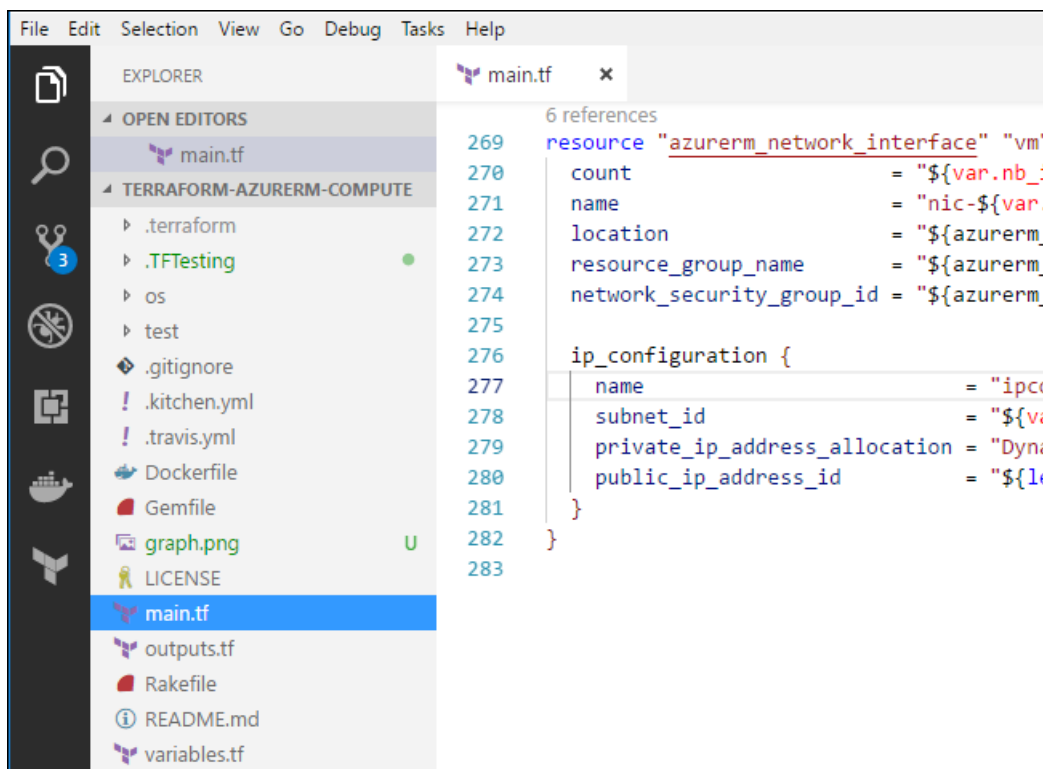


#### NOTE

In this example, our folder was named *terraform-azurerm-compute*.

### Open the folder in Visual Studio Code

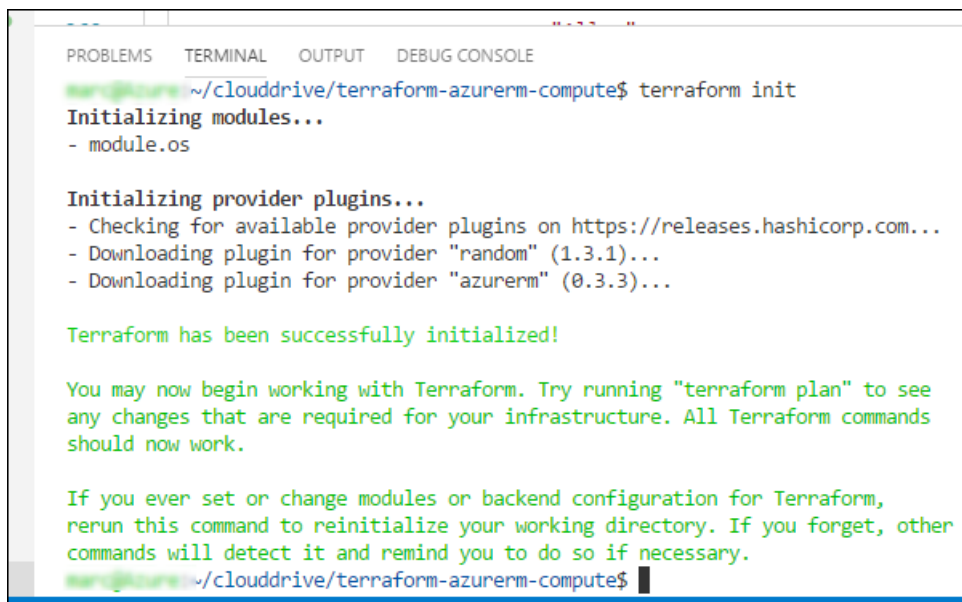
1. Launch Visual Studio Code.
2. From the menu bar, select **File > Open Folder** and navigate to and select the folder you created in the previous step.



### Initialize Terraform

Before you can begin using the Terraform commands from within Visual Studio Code, you download the plug-ins for two Azure providers: random and azurerm.

1. In the Terminal pane of the Visual Studio Code IDE, enter `terraform init`.



```
PROBLEMS  TERMINAL  OUTPUT  DEBUG CONSOLE

~/cloudrive/terraform-azurerm-compute$ terraform init
Initializing modules...
- module.os

Initializing provider plugins...
- Checking for available provider plugins on https://releases.hashicorp.com...
- Downloading plugin for provider "random" (1.3.1)...
- Downloading plugin for provider "azurerm" (0.3.3)...

Terraform has been successfully initialized!

You may now begin working with Terraform. Try running "terraform plan" to see
any changes that are required for your infrastructure. All Terraform commands
should now work.

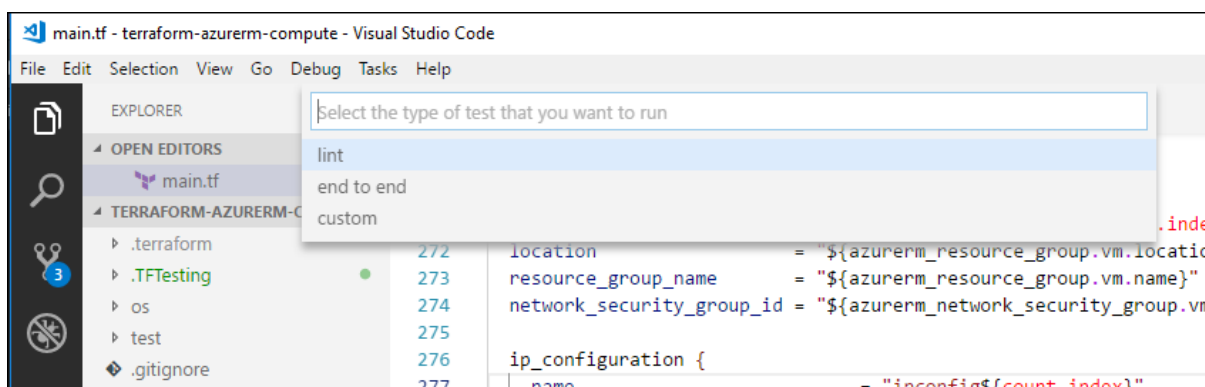
If you ever set or change modules or backend configuration for Terraform,
rerun this command to reinitialize your working directory. If you forget, other
commands will detect it and remind you to do so if necessary.

~/cloudrive/terraform-azurerm-compute$
```

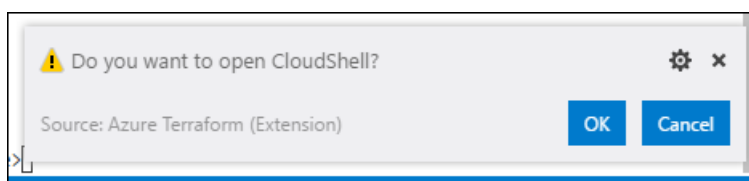
2. Enter `az login`, press <Enter>, and follow the on-screen instructions.

### Module test: *lint*

1. From the menu bar, select **View > Command Palette > Azure Terraform: Execute Test**.
2. From the list of test-type options, select *lint*.



3. When the confirmation appears, select **OK**, and follow the on-screen instructions.



### NOTE

When you execute either the *lint* or *end to end* test, Azure uses a container service to provision a test machine to perform the actual test. For this reason, your test results may typically take several minutes to be returned.

After a few moments, you see a listing in the Terminal pane similar to this example:



```
PROBLEMS  TERMINAL  OUTPUT  DEBUG CONSOLE

Requesting a Cloud Shell...
Connecting terminal...
ms@azure: ~$ cd "$HOME/cloudrive/terraform-azurerm-compute/.TFTesting"
source createacitest.sh && terraform fmt && terraform init && terraform apply -auto-approve && ter
Run the following command to get the logs from the ACI container: az container logs -g tfTestResou
"

ms@azure: ~/cloudrive/terraform-azurerm-compute/.TFTesting$ source createacitest.sh && terraform
ainer_group.TFTest &&
echo "
> Run the following command to get the logs from the ACI container: az container logs -g tfTestRes
> "
testfile.tf

Initializing provider plugins...
- Checking for available provider plugins on https://releases.hashicorp.com...
- Downloading plugin for provider "azurerm" (1.12.0)...

The following providers do not have any version constraints in configuration,
so the latest version was installed.

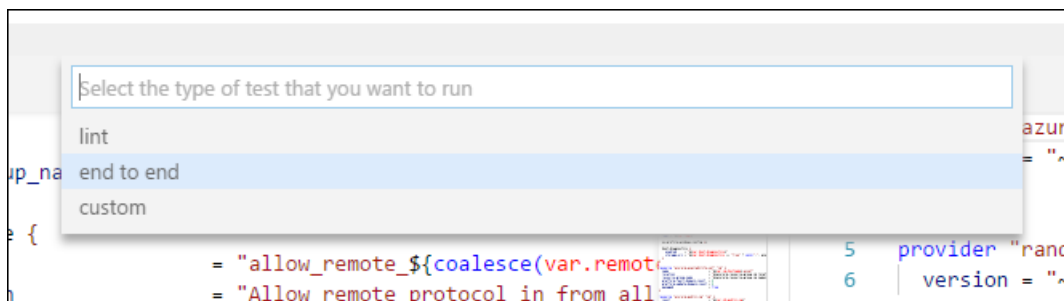
To prevent automatic upgrades to new major versions that may introduce
changes, it is recommended to use specific version constraints to ensure
consistent behavior across environments. To accomplish this, you can use
"version" = "1.12.0" to constrain the azurerm provider to the version
you are currently using.

container.0.volume.0.name:
container.0.volume.0.read_only:
container.0.volume.0.share_name:
container.0.volume.0.storage_account_key:
container.0.volume.0.storage_account_name:
fqdn:
ip_address:
ip_address_type:
location:
name:
os_type:
resource_group_name:
restart_policy:
tags.%:
azurerm_container_group.TFTest: Creation complete after 3s (ID: /subscriptions/9f6b586c-5074-4867-9f
Apply complete! Resources: 2 added, 0 changed, 0 destroyed.
The resource azurerm_container_group.TFTest in the module root has been marked as tainted!

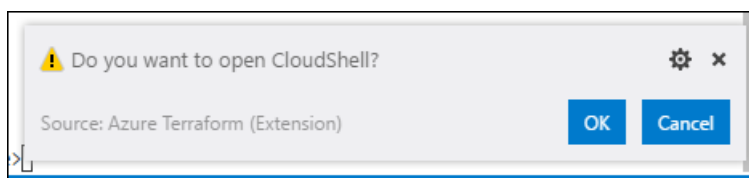
Run the following command to get the logs from the ACI container: az container logs -g tfTestResou
ms@azure: ~/cloudrive/terraform-azurerm-compute/.TFTesting$
```

## Test the module

1. From the menu bar, select **View > Command Palette > Azure Terraform: Execute Test**.
2. From the list of test type options, select **end to end**.



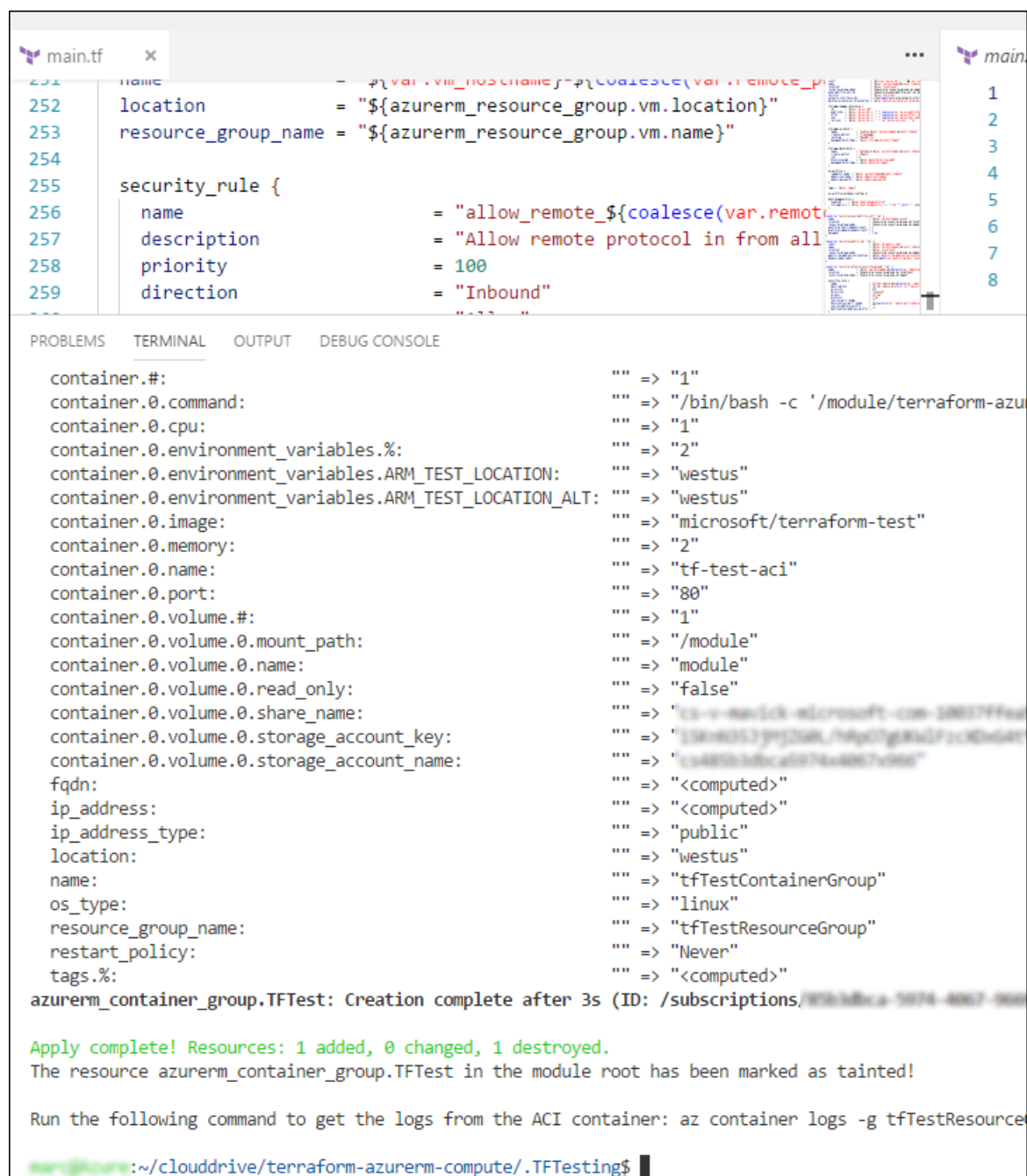
3. When the confirmation appears, select **OK**, and follow the on-screen instructions.



## NOTE

When you execute either the **lint** or **end to end** test, Azure uses a container service to provision a test machine to perform the actual test. For this reason, your test results may typically take several minutes to be returned.

After a few moments, you see a listing in the Terminal pane similar to this example:



The screenshot shows a VS Code editor with a Terraform configuration file named `main.tf`. The configuration defines a security rule for an Azure VM. The terminal pane shows the output of the `terraform apply` command, indicating that the resource `azurerm_container_group.TFTest` was created successfully. The output also includes a warning that the resource has been marked as tainted and a command to view the logs from the ACI container.

```
251 name = "${var.vm_resource_group}/${var.remote_p
252 location = "${azurerm_resource_group.vm.location}"
253 resource_group_name = "${azurerm_resource_group.vm.name}"
254
255 security_rule {
256   name = "allow_remote_${coalesce(var.remote_p
257   description = "Allow remote protocol in from all
258   priority = 100
259   direction = "Inbound"
260 }
```

```
container.0.name: "" => "1"
container.0.command: "" => "/bin/bash -c '/module/terraform-azu
container.0.cpu: "" => "1"
container.0.environment_variables.%: "" => "2"
container.0.environment_variables.ARM_TEST_LOCATION: "" => "westus"
container.0.environment_variables.ARM_TEST_LOCATION_ALT: "" => "westus"
container.0.image: "" => "microsoft/terraform-test"
container.0.memory: "" => "2"
container.0.name: "" => "tf-test-aci"
container.0.port: "" => "80"
container.0.volume.#: "" => "1"
container.0.volume.0.mount_path: "" => "/module"
container.0.volume.0.name: "" => "module"
container.0.volume.0.read_only: "" => "false"
container.0.volume.0.share_name: "" => "cs-v-nwicks-microsoft-com-38837ffea
container.0.volume.0.storage_account_key: "" => "1384053795208148p07gk8dFcc0b0487
container.0.volume.0.storage_account_name: "" => "1384053795208148p07gk8dFcc0b0487
fqdn: "" => "<computed>"
ip_address: "" => "<computed>"
ip_address_type: "" => "public"
location: "" => "westus"
name: "" => "tfTestContainerGroup"
os_type: "" => "linux"
resource_group_name: "" => "tfTestResourceGroup"
restart_policy: "" => "Never"
tags.%: "" => "<computed>"
azurerm_container_group.TFTest: Creation complete after 3s (ID: /subscriptions/87563d6c-a5074-4867-9000

Apply complete! Resources: 1 added, 0 changed, 1 destroyed.
The resource azurerm_container_group.TFTest in the module root has been marked as tainted!

Run the following command to get the logs from the ACI container: az container logs -g tfTestResource

mcs@msrc:~/clouddrive/terraform-azurerm-compute/.TFTesting$
```

## Next steps

[List of the Terraform modules available for Azure \(and other supported providers\)](#)

In this tutorial, you'll learn how to use the combination of [Terraform](#) and [Yeoman](#). Terraform is a tool for creating infrastructure on Azure. Yeoman makes it easy to create Terraform modules.

In this article, you learn how to do the following tasks:

- Create a base Terraform template using the Yeoman module generator.
- Test the Terraform template using two different methods.
- Run the Terraform module using a Docker file.
- Run the Terraform module natively in Azure Cloud Shell.

## Prerequisites

- **Azure subscription:** If you don't have an Azure subscription, create a [free account](#) before you begin.
- **Visual Studio Code:** [Download Visual Studio Code](#) for your platform.
- **Terraform:** [Install Terraform](#) to run the module created by Yeoman.
- **Docker:** [Install Docker](#) to run the module created by the Yeoman generator.
- **Go programming language:** [Install Go](#) as Yeoman-generated test cases are code using the Go language.

### NOTE

Most of the procedures in this tutorial involve the command line interface. The steps described apply to all operating systems and command line tools. For the examples, PowerShell was chosen for the local environment and Git Bash for the cloud shell environment.

## Prepare your environment

### Install Node.js

To use Terraform in the Cloud Shell, you need to [install Node.js](#) 6.0+.

### NOTE

To verify that Node.js is installed, open a terminal window and enter `node --version`.

### Install Yeoman

Run the following command:

```
npm install -g yo
```

```

Yeoman Doctor
Running sanity checks on your system

✓ Global configuration file is valid
✓ NODE_PATH matches the npm root
✓ Node.js version
✓ No .bowerrc file in home directory
✓ No .yo-rc.json file in home directory
✓ npm version
✓ yo version

Everything looks all right!
+ yo@2.0.5
added 537 packages in 141.668s

C:\Users\j-mavick>

```

## Install the Yeoman template for Terraform module

Run the following command:

```
npm install -g generator-az-terra-module
```

```

C:\Users\j-mavick>npm install -g generator-az-terra-module
+ generator-az-terra-module@0.2.1
added 338 packages in 52.171s

C:\Users\j-mavick>

```

To verify that Yeoman is installed, run the following command:

```
yo --version
```

## Create a directory for the Yeoman-generated module

The Yeoman template generates files in the current directory. For this reason, you need to create a directory.

This empty directory is required to be put under \$GOPATH/src. For more information about this path, see the article [Setting GOPATH](#).

1. Navigate to the parent directory from which to create a new directory.
2. Run the following command replacing the placeholder. For this example, a directory name of `GeneratorDocSample` is used.

```
mkdir <new-directory-name>
```

```

PS C:\Users\j-mavick> mkdir GeneratorDocSample

Directory: C:\Users\j-mavick

Mode                LastWriteTime         Length Name
----                -
d-----          8/29/2018   8:25 AM             GeneratorDocSample

```

3. Navigate to the new directory:

```
cd <new-directory-name>
```

```

PS C:\Users\j-mavick> cd GeneratorDocSample
PS C:\Users\j-mavick\GeneratorDocSample> ls
PS C:\Users\j-mavick\GeneratorDocSample>

```

# Create a base module template

1. Run the following command:

```
yo az-terra-module
```

2. Follow the on-screen instructions to provide the following information:

- Terraform module project Name - A value of `doc-sample-module` is used for the example.

```
We're constantly looking for ways to make yo better!
May we anonymously report usage statistics to improve the tool over time?
More info: https://github.com/yeoman/insight & http://yeoman.io
? Terraform module project Name doc-sample-module
```

- Would you like to include the Docker image file? - Enter `y`. If you select `n`, the generated module code will support running only in native mode.

```
We're constantly looking for ways to make yo better!
May we anonymously report usage statistics to improve the tool over time?
More info: https://github.com/yeoman/insight & http://yeoman.io
? Terraform module project Name doc-sample-module
? Would you like to include the Docker image file? (y/N) y
```

3. List the directory contents to view the resulting files that are created:

```
ls
```

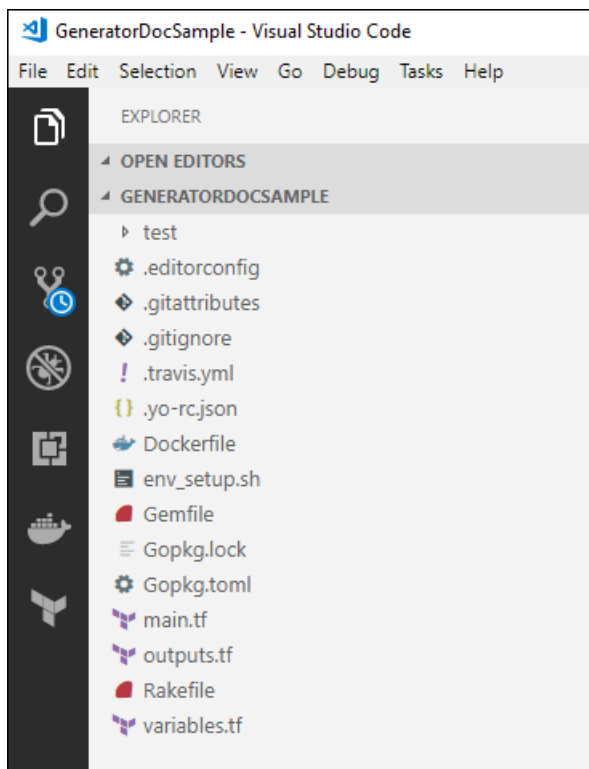
```
PS C:\Users\... \GeneratorDocSample> ls

Directory: C:\Users\... \GeneratorDocSample

Mode                LastWriteTime         Length Name
----                -
d-----          8/29/2018   8:32 AM                test
-a----          8/29/2018   8:32 AM             372 .editorconfig
-a----          8/29/2018   8:32 AM              83 .gitattributes
-a----          8/29/2018   8:32 AM             395 .gitignore
-a----          8/29/2018   8:32 AM             993 .travis.yml
-a----          8/29/2018   8:32 AM              68 .yo-rc.json
-a----          8/29/2018   8:32 AM            1167 Dockerfile
-a----          8/29/2018   8:32 AM           3564 env_setup.sh
-a----          8/29/2018   8:32 AM             168 Gemfile
-a----          8/29/2018   8:32 AM           4090 Gopkg.lock
-a----          8/29/2018   8:32 AM             737 Gopkg.toml
-a----          8/29/2018   8:32 AM             123 main.tf
-a----          8/29/2018   8:32 AM             104 outputs.tf
-a----          8/29/2018   8:32 AM             970 Rakefile
-a----          8/29/2018   8:32 AM             180 variables.tf
```

## Review the generated module code

1. Launch Visual Studio Code
2. From the menu bar, select **File > Open Folder** and select the folder you created.



The following files were created by the Yeoman module generator. For more information about these files and their usage, see [Terratest in Terraform Modules](#).

- `main.tf` - Defines a module called `random-shuffle`. The input is a `string_list`. The output is the count of the permutations.
- `variables.tf` - Defines the input and output variables used by the module.
- `outputs.tf` - Defines what the module outputs. Here, it's the value returned by `random_shuffle`, which is a built-in, Terraform module.
- `Rakefile` - Defines the build steps. These steps include:
  - `build` - Validates the formatting of the `main.tf` file.
  - `unit` - The generated module skeleton doesn't include code for a unit test. If you want to specify a unit test scenario, you would add that code here.
  - `e2e` - Runs an end-to-end test of the module.
- `test`
  - Test cases are written in Go.
  - All codes in `test` are end-to-end tests.
  - End-to-end tests attempt to provision all of the items defined under `fixture`. The results in the `template_output.go` file are compared with the pre-defined expected values.
  - `Gopkg.lock` and `Gopkg.toml`: Defines the dependencies.

## Test your new Terraform module using a Docker file

This section shows how to test a Terraform module using a Docker file.

### NOTE

This example runs the module locally; not on Azure.

### Confirm Docker is installed and running

From a command prompt, enter `docker version`.

```
Windows PowerShell

PS C:\Users\marvick\GeneratorDocSample> docker version
Client:
Version:      18.06.1-ce
API version:  1.38
Go version:   go1.10.3
Git commit:   e68fc7a
Built:        Tue Aug 21 17:21:34 2018
OS/Arch:      windows/amd64
Experimental: false

Server:
Engine:
Version:      18.06.1-ce
API version:  1.38 (minimum version 1.24)
Go version:   go1.10.3
Git commit:   e68fc7a
Built:        Tue Aug 21 17:36:40 2018
OS/Arch:      windows/amd64
Experimental: false
PS C:\Users\marvick\GeneratorDocSample>
```

The resulting output confirms that Docker is installed.

To confirm that Docker is actually running, enter `docker info`.

```
PS C:\Users\marvick\GeneratorDocSample> docker info
Containers: 0
  Running: 0
  Paused: 0
  Stopped: 0
Images: 0
Server Version: 18.06.1-ce
Storage Driver: windowsfilter
Windows:
Logging Driver: json-file
Plugins:
  Volume: local
  Network: ics l2bridge l2tunnel nat null overlay transparent
  Log: awslogs etwlogs fluentd gelf json-file logentries splunk syslog
Swarm: inactive
Default Isolation: hyperv
Kernel Version: 10.0 17134 (17134.1.amd64fre.rs4_release.180410-1804)
Operating System: Windows 10 Enterprise Version 1803 (OS Build 17134.228)
OSType: windows
Architecture: x86_64
CPUs: 4
Total Memory: 15.87GiB
Name: HPE-110117
ID: ZRLC:S2P5:ARXA:7PVM:W7OL:G2Z2:WR5G:LEPH:OXXR:KP5Z:YC2V:6XZT
Docker Root Dir: C:\ProgramData\Docker
Debug Mode (client): false
Debug Mode (server): true
  File Descriptors: -1
  Goroutines: 26
  System Time: 2018-09-04T14:40:01.0579354-07:00
  EventsListeners: 1
Registry: https://index.docker.io/v1/
Labels:
Experimental: false
Insecure Registries:
  127.0.0.0/8
Live Restore Enabled: false
PS C:\Users\marvick\GeneratorDocSample>
```

## Set up a Docker container

1. From a command prompt, enter

```
docker build --build-arg BUILD_ARM_SUBSCRIPTION_ID= --build-arg BUILD_ARM_CLIENT_ID= --build-arg BUILD_ARM_CLIENT_SECRET= --build-arg BUILD_ARM_TENANT_ID= -t terra-mod-example .
```

The message **Successfully built** will be displayed.

```
Successfully built 2ca48eb1668c
Successfully tagged terra-mod-example:latest
```

2. From the command prompt, enter `docker image ls` to see your created module `terra-mod-example` listed.

REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
terra-mod-example	latest	2ca48eb1668c	41 seconds ago	1.5808

3. Enter `docker run -it terra-mod-example /bin/sh`. After running the `docker run` command, you're in the Docker environment. At that point, you can discover the file by using the `ls` command.

```
PS D:\Tryout\GeneratorDocSample> docker run -it terra-mod-example /bin/sh
# ls
Dockerfile  Gemfile.lock  Gopkg.toml  env_setup.sh  outputs.tf  variables.tf
Gemfile     Gopkg.lock   Rakefile    main.tf       test
```

## Build the module

1. Run the following command:

```
bundle install
```

2. Run the following command:

```
rake build
```

```
# rake build
Using dep ensure to install required go packages.
INFO: Styling Terraform configurations...
INFO: Done!
INFO: Linting Terraform configurations...
INFO: Done!
```

## Run the end-to-end test

1. Run the following command:

```
rake e2e
```

2. After a few moments, the **PASS** message will appear.

```
PASS
ok      terraform-azurerem-template/test 0.908s
#
```

3. Enter `exit` to complete the test and exit the Docker environment.

# Use Yeoman generator to create and test a module in Cloud Shell

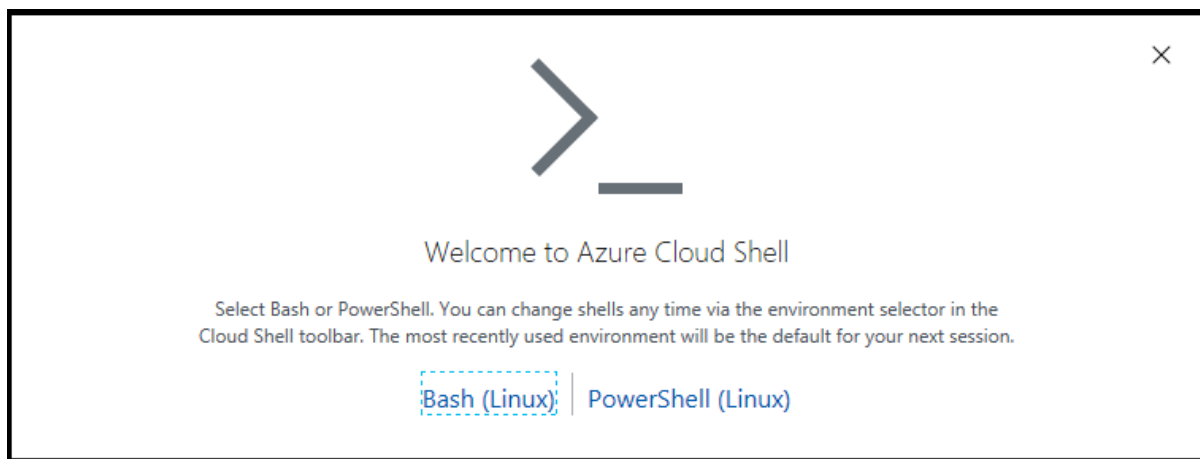
In this section, the Yeoman generator is used to create and test a module in Cloud Shell. Using Cloud Shell instead of using a Docker file greatly simplifies the process. Using Cloud Shell, the following products are all pre-installed:

- Node.js
- Yeoman
- Terraform

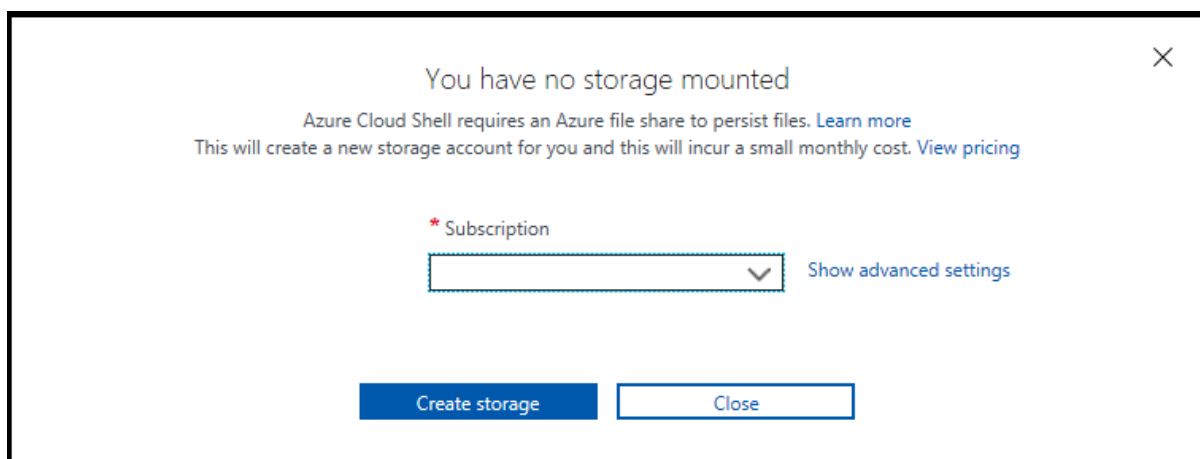
## Start a Cloud Shell session

1. Start an Azure Cloud Shell session via either the [Azure portal](#), [shell.azure.com](https://shell.azure.com), or the [Azure mobile app](#).
2. The **Welcome to Azure Cloud Shell** page opens. Select **Bash (Linux)**.





3. If you have not already set up an Azure storage account, the following screen appears. Select **Create storage**.



4. Azure Cloud Shell launches in the shell you previously selected and displays information for the cloud drive it just created for you.

```

Azure Cloud Shell
Bash | [Power] [Help] [Settings] [File] [Terminal] [Full Screen]
Your cloud drive has been created in:
Subscription Id: 85b3dbca-5974-4067-9669-67a141095a76
Resource group: cloud-shell-storage-westus
Storage account: cs485b3dbca5974x4067x966
File share: cs-v-mavick-microsoft-com-10037ffea5c3e9a7
Initializing your account for Cloud Shell...-
Requesting a Cloud Shell.Succeeded.
Connecting terminal...
Welcome to Azure Cloud Shell
Type "az" to use Azure CLI 2.0
Type "help" to learn about Cloud Shell
marc@Azure:~$

```

### Prepare a directory to hold your Terraform module

1. At this point, Cloud Shell will have already configured GOPATH in your environment variables for you. To see the path, enter `go env`.
2. Create the \$GOPATH directory, if one doesn't already exist: Enter `mkdir ~/go`.
3. Create a directory within the \$GOPATH directory. This directory is used to hold the different project

directories created in this example.

```
mkdir ~/go/src
```

4. Create a directory to hold your Terraform module replacing the placeholder. For this example, a directory name of `my-module-name` is used.

```
mkdir ~/go/src/<your-module-name>
```

5. Navigate to your module directory:

```
cd ~/go/src/<your-module-name>
```

### Create and test your Terraform module

1. Run the following command and follow the instructions. When asked if you want to create the Docker files, you enter `N`.

```
yo az-terra-module
```

2. Run the following command to install the dependencies:

```
bundle install
```

3. Run the following command to build the module:

```
rake build
```

```
# rake build
Using dep ensure to install required go packages.
INFO: Styling Terraform configurations...
INFO: Done!
INFO: Linting Terraform configurations...
INFO: Done!
```

4. Run the following command to run the test:

```
rake e2e
```

```
PASS
ok      terraform-azurerem-tesplate/test 0.900s
#
```

## Next steps

[Install and use the Azure Terraform Visual Studio Code extension.](#)

#### NOTE

The sample code in this article does not work with version 0.12 (and greater).

You can use Azure Terraform modules to create reusable, composable, and testable components. Terraform modules incorporate encapsulation that's useful in implementing infrastructure as code processes.

It's important to implement quality assurance when you create Terraform modules. Unfortunately, limited documentation is available to explain how to author unit tests and integration tests in Terraform modules. This tutorial introduces a testing infrastructure and best practices that we adopted when we built our [Azure Terraform modules](#).

We looked at all the most popular testing infrastructures and chose [Terratest](#) to use for testing our Terraform modules. Terratest is implemented as a Go library. Terratest provides a collection of helper functions and patterns for common infrastructure testing tasks, like making HTTP requests and using SSH to access a specific virtual machine. The following list describes some of the major advantages of using Terratest:

- **It provides convenient helpers to check infrastructure.** This feature is useful when you want to verify your real infrastructure in the real environment.
- **The folder structure is clearly organized.** Your test cases are organized clearly and follow the [standard Terraform module folder structure](#).
- **All test cases are written in Go.** Most developers who use Terraform are Go developers. If you're a Go developer, you don't have to learn another programming language to use Terratest. Also, the only dependencies that are required for you to run test cases in Terratest are Go and Terraform.
- **The infrastructure is highly extensible.** You can extend additional functions on top of Terratest, including Azure-specific features.

## Prerequisites

This hands-on article is platform-independent. You can run the code examples that we use in this article on Windows, Linux, or MacOS.

Before you begin, install the following software:

- **Go programming language:** Terraform test cases are written in [Go](#).
- **dep:** [dep](#) is a dependency management tool for Go.
- **Azure CLI:** The [Azure CLI](#) is a command-line tool you can use to manage Azure resources. (Terraform supports authenticating to Azure through a service principal or [via the Azure CLI](#).)
- **mage:** We use the [mage executable](#) to show you how to simplify running Terratest cases.

## Create a static webpage module

In this tutorial, you create a Terraform module that provisions a static webpage by uploading a single HTML file to an Azure Storage blob. This module gives users from around the world access to the webpage through a URL that the module returns.

## NOTE

Create all files that are described in this section under your `GOPATH` location.

First, create a new folder named `staticwebpage` under your GoPath `src` folder. The overall folder structure of this tutorial is shown in the following example. Files marked with an asterisk `(*)` are the primary focus in this section.

```
GoPath/src/staticwebpage
├── examples
│   └── hello-world
│       ├── index.html
│       └── main.tf
├── test
│   ├── fixtures
│   │   └── storage-account-name
│   │       ├── empty.html
│   │       └── main.tf
│   ├── hello_world_example_test.go
│   └── storage_account_name_unit_test.go
├── main.tf (*)
├── outputs.tf (*)
└── variables.tf (*)
```

The static webpage module accepts three inputs. The inputs are declared in `./variables.tf`:

```
variable "location" {
  description = "The Azure region in which to create all resources."
}

variable "website_name" {
  description = "The website name to use to create related resources in Azure."
}

variable "html_path" {
  description = "The file path of the static home page HTML in your local file system."
  default    = "index.html"
}
```

As we mentioned earlier in the article, this module also outputs a URL that's declared in `./outputs.tf`:

```
output "homepage_url" {
  value = azurerm_storage_blob.homepage.url
}
```

The main logic of the module provisions four resources:

- **resource group:** The name of the resource group is the `website_name` input appended by `-staging-rg`.
- **storage account:** The name of the storage account is the `website_name` input appended by `data001`. To adhere to the name limitations of the storage account, the module removes all special characters and uses lowercase letters in the entire storage account name.
- **fixed name container:** The container is named `wwwroot` and is created in the storage account.
- **single HTML file:** The HTML file is read from the `html_path` input and uploaded to `wwwroot/index.html`.

The static webpage module logic is implemented in `./main.tf`:

```

resource "azurerm_resource_group" "main" {
  name      = "${var.website_name}-staging-rg"
  location  = var.location
}

resource "azurerm_storage_account" "main" {
  name                        = "${lower(replace(var.website_name, "[[:^alnum:]]/", ""))}data001"
  resource_group_name        = azurerm_resource_group.main.name
  location                   = azurerm_resource_group.main.location
  account_tier               = "Standard"
  account_replication_type   = "LRS"
}

resource "azurerm_storage_container" "main" {
  name                = "wwwroot"
  resource_group_name = azurerm_resource_group.main.name
  storage_account_name = azurerm_storage_account.main.name
  container_access_type = "blob"
}

resource "azurerm_storage_blob" "homepage" {
  name            = "index.html"
  resource_group_name = azurerm_resource_group.main.name
  storage_account_name = azurerm_storage_account.main.name
  storage_container_name = azurerm_storage_container.main.name
  source          = var.html_path
  type            = "block"
  content_type     = "text/html"
}

```

## Unit test

Terratest is designed for integration tests. For that purpose, Terratest provisions real resources in a real environment. Sometimes, integration test jobs can become exceptionally large, especially when you have a large number of resources to provision. The logic that converts storage account names that we refer to in the preceding section is a good example.

But, we don't really need to provision any resources. We only want to make sure that the naming conversion logic is correct. Thanks to the flexibility of Terratest, we can use unit tests. Unit tests are local running test cases (although internet access is required). Unit test cases execute `terraform init` and `terraform plan` commands to parse the output of `terraform plan` and look for the attribute values to compare.

The rest of this section describes how we use Terratest to implement a unit test to make sure that the logic used to convert storage account names is correct. We are interested only in the files marked with an asterisk `(*)`.

```

GoPath/src/staticwebpage
├── examples
│   └── hello-world
│       ├── index.html
│       └── main.tf
├── test
│   ├── fixtures
│   │   └── storage-account-name
│   │       ├── empty.html (*)
│   │       └── main.tf (*)
│   ├── hello_world_example_test.go
│   └── storage_account_name_unit_test.go (*)
├── main.tf
├── outputs.tf
└── variables.tf

```

First, we use an empty HTML file named `./test/fixtures/storage-account-name/empty.html` as a placeholder.

The file `./test/fixtures/storage-account-name/main.tf` is the test case frame. It accepts one input, `website_name`, which is also the input of the unit tests. The logic is shown here:

```
variable "website_name" {
  description = "The name of your static website."
}

module "staticwebpage" {
  source      = "../../.."
  location    = "West US"
  website_name = var.website_name
  html_path   = "empty.html"
}
```

The major component is the implementation of the unit tests in `./test/storage_account_name_unit_test.go`.

Go developers probably will notice that the unit test matches the signature of a classic Go test function by accepting an argument of type `*testing.T`.

In the body of the unit test, we have a total of five cases that are defined in variable `testCases` ( `key` as input, and `value` as expected output). For each unit test case, we first run `terraform init` and target the test fixture folder ( `./test/fixtures/storage-account-name/` ).

Next, a `terraform plan` command that uses specific test case input (take a look at the `website_name` definition in `tfOptions` ) saves the result to `./test/fixtures/storage-account-name/terraform.tfplan` (not listed in the overall folder structure).

This result file is parsed to a code-readable structure by using the official Terraform plan parser.

Now, we look for the attributes we're interested in (in this case, the `name` of the `azurerm_storage_account` ) and compare the results with the expected output:

```

package test

import (
    "os"
    "path"
    "testing"

    "github.com/gruntwork-io/terratest/modules/terraform"
    terraformCore "github.com/hashicorp/terraform/terraform"
)

func TestUT_StorageAccountName(t *testing.T) {
    t.Parallel()

    // Test cases for storage account name conversion logic
    testCases := map[string]string{
        "TestWebsiteName": "testwebsitenamedata001",
        "ALLCAPS":          "allcapsdata001",
        "S_p-e(c)i.a_l":    "specialdata001",
        "A1phaNum321":       "a1phanum321data001",
        "E5e-y7h_ng":        "e5ey7hngdata001",
    }

    for input, expected := range testCases {
        // Specify the test case folder and "-var" options
        tfOptions := &terraform.Options{
            TerraformDir: "./fixtures/storage-account-name",
            Vars: map[string]interface{}{
                "website_name": input,
            },
        },
    }

    // Terraform init and plan only
    tfPlanOutput := "terraform.tfplan"
    terraform.Init(t, tfOptions)
    terraform.RunTerraformCommand(t, tfOptions, terraform.FormatArgs(tfOptions, "plan", "-out="+tfPlanOutput)...)

    // Read and parse the plan output
    f, err := os.Open(path.Join(tfOptions.TerraformDir, tfPlanOutput))
    if err != nil {
        t.Fatal(err)
    }
    defer f.Close()
    plan, err := terraformCore.ReadPlan(f)
    if err != nil {
        t.Fatal(err)
    }

    // Validate the test result
    for _, mod := range plan.Diff.Modules {
        if len(mod.Path) == 2 && mod.Path[0] == "root" && mod.Path[1] == "staticwebpage" {
            actual := mod.Resources["azurerm_storage_account.main"].Attributes["name"].New
            if actual != expected {
                t.Fatalf("Expect %v, but found %v", expected, actual)
            }
        }
    }
}

```

To run the unit tests, complete the following steps on the command line:

```
az login    # Required when no service principal environment variables are present
```

```

cd [Your GoPath]/src/staticwebpage
dep init    # Run only once for this folder
dep ensure # Required to run if you imported new packages in test cases
cd test
go fmt
go test -run TestUT_StorageAccountName

```

The traditional Go test result returns in about a minute.

## Integration test

In contrast to unit tests, integration tests must provision resources to a real environment for an end-to-end perspective. Terratest does a good job with this kind of task.

Best practices for Terraform modules include installing the `examples` folder. The `examples` folder contains some end-to-end samples. To avoid working with real data, why not test those samples as integration tests? In this section, we focus on the three files that are marked with an asterisk `(*)` in the following folder structure:

```

GoPath/src/staticwebpage
├── examples
│   └── hello-world
│       ├── index.html (*)
│       └── main.tf      (*)
├── test
│   ├── fixtures
│   │   └── storage-account-name
│   │       ├── empty.html
│   │       └── main.tf
│   ├── hello_world_example_test.go (*)
│   └── storage_account_name_unit_test.go
├── main.tf
├── outputs.tf
└── variables.tf

```

Let's start with the samples. A new sample folder named `hello-world/` is created in the `./examples/` folder. Here, we provide a simple HTML page to be uploaded: `./examples/hello-world/index.html`.

```

<!DOCTYPE html>
<html>
<head>
  <meta charset="utf-8">
  <title>Hello World</title>
</head>
<body>
  <h1>Hi, Terraform Module</h1>
  <p>This is a sample web page to demonstrate Terratest.</p>
</body>
</html>

```

The Terraform sample `./examples/hello-world/main.tf` is similar to the one shown in the unit test. There's one significant difference: the sample also prints out the URL of the uploaded HTML as a webpage named `homepage`.



```

variable "website_name" {
  description = "The name of your static website."
  default     = "Hello-World"
}

module "staticwebpage" {
  source      = "../.."
  location    = "West US"
  website_name = var.website_name
}

output "homepage" {
  value = module.staticwebpage.homepage_url
}

```

We use Terratest and classic Go test functions again in the integration test file `./test/hello_world_example_test.go`.

Unlike unit tests, integration tests create actual resources in Azure. That's why you need to be careful to avoid naming conflicts. (Pay special attention to some globally unique names like storage account names.) Therefore, the first step of the testing logic is to generate a randomized `websiteName` by using the `UniqueId()` function provided by Terratest. This function generates a random name that has lowercase letters, uppercase letters, or numbers.

`tfOptions` makes all Terraform commands that target the `./examples/hello-world/` folder. It also makes sure that `website_name` is set to the randomized `websiteName`.

Then, `terraform init`, `terraform apply`, and `terraform output` are executed, one by one. We use another helper function, `HttpGetWithCustomValidation()`, which is provided by Terratest. We use the helper function to make sure that HTML is uploaded to the output `homepage` URL that's returned by `terraform output`. We compare the HTTP GET status code with `200` and look for some keywords in the HTML content. Finally, `terraform destroy` is "promised" to be executed by leveraging the `defer` feature of Go.

```

package test

import (
    "fmt"
    "strings"
    "testing"

    "github.com/gruntwork-io/terratest/modules/http-helper"
    "github.com/gruntwork-io/terratest/modules/random"
    "github.com/gruntwork-io/terratest/modules/terraform"
)

func TestIT_HelloWorldExample(t *testing.T) {
    t.Parallel()

    // Generate a random website name to prevent a naming conflict
    uniqueID := random.UniqueId()
    websiteName := fmt.Sprintf("Hello-World-%s", uniqueID)

    // Specify the test case folder and "-var" options
    tfOptions := &terraform.Options{
        TerraformDir: "../examples/hello-world",
        Vars: map[string]interface{}{
            "website_name": websiteName,
        },
    }

    // Terraform init, apply, output, and destroy
    defer terraform.Destroy(t, tfOptions)
    terraform.InitAndApply(t, tfOptions)
    homepage := terraform.Output(t, tfOptions, "homepage")

    // Validate the provisioned webpage
    http_helper.HttpGetWithCustomValidation(t, homepage, func(status int, content string) bool {
        return status == 200 &&
            strings.Contains(content, "Hi, Terraform Module") &&
            strings.Contains(content, "This is a sample web page to demonstrate Terratest.")
    })
}

```

To run the integration tests, complete the following steps on the command line:

```
az login    # Required when no service principal environment variables are present
```

```

cd [Your GoPath]/src/staticwebpage
dep init    # Run only once for this folder
dep ensure  # Required to run if you imported new packages in test cases
cd test
go fmt
go test -run TestIT_HelloWorldExample

```

The traditional Go test result returns in about two minutes. You could also run both unit tests and integration tests by executing these commands:

```

go fmt
go test

```

Integration tests take much longer than unit tests (two minutes for one integration case compared to one minute for five unit cases). But it's your decision whether to use unit tests or integration tests in a scenario. Typically, we prefer to use unit tests for complex logic by using Terraform HCL functions. We usually use integration tests for the

end-to-end perspective of a user.

## Use mage to simplify running Terratest cases

Running test cases in Azure Cloud Shell requires executing different commands in various directories. To make this process more efficient, we introduce the build system in our project. In this section, we use a Go build system, `mage`, for the job.

The only thing required by `mage` is `magefile.go` in your project's root directory (marked with `(+)` in the following example):

```
GoPath/src/staticwebpage
├── examples
│   └── hello-world
│       ├── index.html
│       └── main.tf
├── test
│   ├── fixtures
│   │   └── storage-account-name
│   │       ├── empty.html
│   │       └── main.tf
│   ├── hello_world_example_test.go
│   └── storage_account_name_unit_test.go
├── magefile.go (+)
├── main.tf
├── outputs.tf
└── variables.tf
```

Here's an example of `./magefile.go`. In this build script, written in Go, we implement five build steps:

- Clean**: The step removes all generated and temporary files that are generated during test executions.
- Format**: The step runs `terraform fmt` and `go fmt` to format your code base.
- Unit**: The step runs all unit tests (by using the function name convention `TestUT_*`) under the `./test/` folder.
- Integration**: The step is similar to **Unit**, but instead of unit tests, it executes integration tests (`TestIT_*`).
- Full**: The step runs **Clean**, **Format**, **Unit**, and **Integration** in sequence.

```
// +build mage

// Build a script to format and run tests of a Terraform module project
package main

import (
    "fmt"
    "os"
    "path/filepath"

    "github.com/magefile/mage/mg"
    "github.com/magefile/mage/sh"
)

// The default target when the command executes `mage` in Cloud Shell
var Default = Full

// A build step that runs Clean, Format, Unit and Integration in sequence
func Full() {
    mg.Deps(Unit)
    mg.Deps(Integration)
}

// A build step that runs unit tests
func Unit() error {
    mg.Deps(Clean)
```

```

mg.Deps(Clean)
mg.Deps(Format)
fmt.Println("Running unit tests...")
return sh.RunV("go", "test", "./test/", "-run", "TestUT_", "-v")
}

// A build step that runs integration tests
func Integration() error {
mg.Deps(Clean)
mg.Deps(Format)
fmt.Println("Running integration tests...")
return sh.RunV("go", "test", "./test/", "-run", "TestIT_", "-v")
}

// A build step that formats both Terraform code and Go code
func Format() error {
fmt.Println("Formatting...")
if err := sh.RunV("terraform", "fmt", "."); err != nil {
return err
}
return sh.RunV("go", "fmt", "./test/")
}

// A build step that removes temporary build and test files
func Clean() error {
fmt.Println("Cleaning...")
return filepath.Walk(".", func(path string, info os.FileInfo, err error) error {
if err != nil {
return err
}
if info.IsDir() && info.Name() == "vendor" {
return filepath.SkipDir
}
if info.IsDir() && info.Name() == ".terraform" {
os.RemoveAll(path)
fmt.Printf("Removed \"%v\"\n", path)
return filepath.SkipDir
}
if !info.IsDir() && (info.Name() == "terraform.tfstate" ||
info.Name() == "terraform.tfplan" ||
info.Name() == "terraform.tfstate.backup") {
os.Remove(path)
fmt.Printf("Removed \"%v\"\n", path)
}
return nil
})
}

```

You can use the following commands to execute a full test suite. The code is similar to the running steps we used in an earlier section.

```
az login    # Required when no service principal environment variables are present
```

```

cd [Your GoPath]/src/staticwebpage
dep init    # Run only once for this folder
dep ensure  # Required to run if you imported new packages in magefile or test cases
go fmt      # Only required when you change the magefile
mage

```

You can replace the last command line with additional mage steps. For example, you can use `mage unit` or `mage clean`. It's a good idea to embed `dep` commands and `az login` in the magefile. We don't show the code here.

With mage, you could also share the steps by using the Go package system. In that case, you can simplify magefiles across all your modules by referencing only a common implementation and declaring dependencies (`mg.Deps()`).

### Optional: Set service principal environment variables to run acceptance tests

Instead of executing `az login` before tests, you can complete Azure authentication by setting the service principal environment variables. Terraform publishes a [list of environment variable names](#). (Only the first four of these environment variables are required.) Terraform also publishes detailed instructions that explain how to [obtain the value of these environment variables](#).

## Next steps

[Terratest GitHub page](#).