Perform Dynamic Operations with Functions

developer.hashicorp.com/terraform/tutorials/certification-associate-tutorials/functions

The Terraform configuration language allows you to write declarative expressions to create infrastructure. While the configuration language is not a programming language, you can use several built-in functions to perform operations dynamically.

In this tutorial, you will:

- use the `templatefile` function to dynamically create an EC2 instance user data script.
- use the `lookup` function to reference values from a map.
- use the `file` function to read the contents of a file.

Prerequisites

You can complete this tutorial using the same workflow with either Terraform OSS or Terraform Cloud. Terraform Cloud is a platform that you can use to manage and execute your Terraform projects. It includes features like remote state and execution, structured plan output, workspace resource summaries, and more.

Select the **Terraform Cloud** tab to complete this tutorial using Terraform Cloud.

This tutorial assumes that you are familiar with the Terraform workflow. If you are new to Terraform, complete the [Get Started tutorials(/collections/terraform/aws-get-started) first.

In order to complete this tutorial, you will need the following:

- Terraform v1.2+ installed locally.
- An AWS account with local credentials configured for use with Terraform.

This tutorial assumes that you are familiar with the Terraform and Terraform Cloud workflows. If you are new to Terraform, complete the [Get Started tutorials(/collections/terraform/aws-get-started) first. If you are new to Terraform Cloud, complete the Terraform Cloud Get Started tutorials first.

In order to complete this tutorial, you will need the following:

- Terraform v1.2+ installed locally.
- An AWS account.
- A Terraform Cloud account with Terraform Cloud locally authenticated.
- A Terraform Cloud variable set configured with your AWS credentials.

Clone the example repository

Clone the Learn Terraform Functions example repository. This repository contains example configuration for you to use to practice using functions to create dynamic EC2 configuration.

```
$ git clone https://github.com/hashicorp/learn-terraform-functions.git
```

Navigate to the repository directory in your terminal.

```
$ cd learn-terraform-functions
```

Use `templatefile` to dynamically generate a script

AWS lets you configure EC2 instances to run a user-provided script -- called a user-data script -- at boot time. You can use Terraform's `templatefile` function to interpolate values into the script at resource creation time. This makes the script more adaptable and re-usable.

Open the `user_data.tftpl` file, which will be the user data script for your EC2 instance. This template file is a shell script to configure and deploy an application. Notice the `${department}` and `${name}` references -- Terraform will interpolate these values using the `templatefile` function.

user_data.tftpl

```
#!/bin/bash

# Install necessary dependencies
sudo DEBIAN_FRONTEND=noninteractive apt-get -y -o Dpkg::Options::="--force-
confdef" -o Dpkg::Options::="--force-confold" dist-upgrade
sudo apt-get -y -qq install curl wget git vim apt-transport-https ca-certificates
sudo add-apt-repository ppa:longsleep/golang-backports -y
sudo apt -y -qq install golang-go

# Setup sudo to allow no-password sudo for your group and adding your user
sudo groupadd -r ${department}
sudo useradd -m -s /bin/bash ${name}
sudo usermod -a -G ${department} ${name}
sudo cp /etc/sudoers /etc/sudoers.orig
echo "${name} ALL=(ALL) NOPASSWD:ALL" | sudo tee /etc/sudoers.d/${name}

# Create GOPATH for your user & download the webapp from github
sudo -H -i -u ${name} -- env bash << EOF
cd /home/${name}
export GOROOT=/usr/lib/go
export GOPATH=/home/${name}/go
export PATH=$PATH:$GOROOT/bin:$GOPATH/bin
go get -d github.com/hashicorp/learn-go-webapp-demo
cd go/src/github.com/hashicorp/learn-go-webapp-demo
go run webapp.go
EOF
```

Next, open the `variables.tf` file. This file includes definitions for the `user_name` and `user_department` input variables, which the configuration uses to set the values for the corresponding template file keys.

variables.tf

```
variable "user_name" {
  description = "The user creating this infrastructure"
  default     = "terraform"
}

variable "user_department" {
  description = "The organization the user belongs to: dev, prod, qa"
  default     = "learn"
}
```

Now open `main.tf`. Add the `user_data` attribute to the `aws_instance` resource block as shown below. The `templatefile` function takes two arguments: the template file name and a map of template value assignments.

```
resource "aws_instance" "web" {
  ami                         = data.aws_ami.ubuntu.id
  instance_type               = "t2.micro"
  subnet_id                   = aws_subnet.subnet_public.id
  vpc_security_group_ids      = [aws_security_group.sg_8080.id]
  associate_public_ip_address = true
  user_data                   = templatefile("user_data.tftpl", { department =
var.user_department, name = var.user_name })
}
```

Save your changes.

Create infrastructure

Initialize this configuration.

```
$ terraform init
Initializing the backend...
##...
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.
```

Open your `terraform.tf` file and uncomment the `cloud` block. Replace the `organization` name with your own Terraform Cloud organization.

terraform.tf

```
terraform {
  cloud {
    organization = "organization-name"
    workspaces {
      name = "learn-terraform-functions"
    }
  }
  required_providers {
    aws = {
      source  = "hashicorp/aws"
      version = "~> 4.4.0"
    }
  }
  required_version = ">= 1.2"
}
```

Initialize your configuration. Terraform will automatically create the `learn-terraform-functions` workspace in your Terraform Cloud organization.

```
$ terraform init
Initializing Terraform Cloud...
Initializing provider plugins...
- Reusing previous version of hashicorp/aws from the dependency lock file
- Installing hashicorp/aws v4.4.0...
- Installed hashicorp/aws v4.4.0 (signed by HashiCorp)
Terraform Cloud has been successfully initialized!
You may now begin working with Terraform Cloud. Try running "terraform plan" to
see any changes that are required for your infrastructure.
If you ever set or change modules or Terraform Settings, run "terraform init"
again to reinitialize your working directory.
```

Note: This tutorial assumes that you are using a tutorial-specific Terraform Cloud organization with a global variable set of your AWS credentials. Review the Create a Credential Variable Set for detailed guidance. If you are using a scoped variable set, assign it to your new workspace now.

Apply your configuration. Respond `yes` to confirm the operation.

```
$ terraform apply
Terraform used the selected providers to generate the following execution
plan. Resource actions are indicated with the following symbols:
  + create

Terraform will perform the following actions:
##...
Plan: 7 to add, 0 to change, 0 to destroy.

Changes to Outputs:
  + web_public_address = (known after apply)
  + web_public_ip      = (known after apply)


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
##...
Apply complete! Resources: 7 added, 0 changed, 0 destroyed.

Outputs:

web_public_address = "3.235.192.120:8080"
web_public_ip = "3.235.192.120"
```

Terraform provisions your network configuration, instance, and provisioning script necessary to launch the example web app. Your `web_public_address` output in your terminal is the address of your web app instance. Navigate to that address in your web browser to verify your configuration.

Destroy your infrastructure before moving to the next section.

```
$ terraform destroy
##...
Terraform used the selected providers to generate the following execution
plan. Resource actions are indicated with the following symbols:
  - destroy

Terraform will perform the following actions:
##...
Plan: 0 to add, 0 to change, 7 to destroy.

Changes to Outputs:
  - ami_value          = "ami-0739f8cdb239fe9ae" -> null
  - web_public_address = "3.235.192.120:8080" -> null
  - web_public_ip      = "3.235.192.120" -> null

Do you really want to destroy all resources?
  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
##...
Apply complete! Resources: 0 added, 0 changed, 7 destroyed.
```

Enter `yes` when prompted to accept your changes.

Use `lookup` function to select AMI

The `lookup` function retrieves the value of a single element from a map, given its key.

Add the following configuration to your `variables.tf` file to declare a new input variable.

variables.tf

```
variable "aws_amis" {
  type = map
  default = {
    "us-east-1" = "ami-0739f8cdb239fe9ae"
    "us-west-2" = "ami-008b09448b998a562"
    "us-east-2" = "ami-0ebc8f6f580a04647"
  }
}
```

This input variable includes a default value of a map of region-specific AMI IDs for three regions.

Now, open the `main.tf` file and remove the data source for your AMI ID.

main.tf

```
- data "aws_ami" "ubuntu" {
- most_recent = true

- filter {
-  name = "name"
-  values = ["ubuntu/images/hvm-ssd/ubuntu-xenial-16.04-amd64-server-*"]
- }

- filter {
-   name = "virtualization-type"
-   values = ["hvm"]
-  }
- owners = ["099720109477"] # Canonical
- }
```

In your `aws_instance` resource, update the `ami` attribute to use the `lookup` function.

main.tf

```
##...
resource "aws_instance" "web" {
- ami                          = data.aws_ami.ubuntu.id
+ ami                          = lookup(var.aws_amis, var.aws_region)
  instance_type                = "t2.micro"
  subnet_id                    = aws_subnet.subnet_public.id
  vpc_security_group_ids       = [aws_security_group.sg_8080.id]
  associate_public_ip_address = true
  user_data                    = templatefile("user_data.tftpl", { department =
var.user_department, name = var.user_name })
}
```

The `ami` is a required attribute for the `aws_instance` resource, so the `lookup` function must return a valid value for Terraform to apply your configuration. The `lookup` function arguments are a map, the key to access in the map, and an optional default value in case the key does not exist.

Next, add the following configuration for an `ami_value` output to your `outputs.tf` file. This output lets you verify the AMI returned by the `lookup` function.

outputs.tf

```
output "ami_value" {
  value = lookup(var.aws_amis, var.aws_region)
}
```

Now run `terraform plan` to review the execution plan for these changes, using a command-line variable flag to set the region to `us-east-2.` The output now includes the selected AMI ID, which Terraform determined using the `lookup` function.

```
$ terraform plan -var "aws_region=us-east-2"
##...
Plan: 7 to add, 0 to change, 0 to destroy.

Changes to Outputs:
  + ami_value          = "ami-0ebc8f6f580a04647"
  + web_public_address = (known after apply)
  + web_public_ip      = (known after apply)
```

Use the `file` function

In this section, you will create a new security group to allow SSH ingress traffic to your instance and configure the instance with an SSH key.

Create an SSH key and a security group resource

Create a local SSH key to pair with the new instance you create so that you can connect securely to your instance.

Generate a new SSH key called `ssh-key`. The argument provided with the `-f` flag creates the key in the current directory and creates two files called `ssh_key` and `ssh_key.pub`. Change the placeholder email address to your email address.

```
$ ssh-keygen -C "your_email@example.com" -f ssh_key
```

When prompted, press enter to leave the passphrase blank on this key.

If you're on a Windows machine use Putty to generate SSH keys by following the instructions here.

Next, add the following configuration to `main.tf` to create a new security group and AWS key pair.

In `main.tf`, add a new `aws_security_group` resource. Copy and append the resource block below to your `main.tf` file.

main.tf

```
resource "aws_security_group" "sg_22" {
  name = "sg_22"
  vpc_id = aws_vpc.vpc.id

  ingress {
    from_port = 22
    to_port   = 22
    protocol  = "tcp"
    cidr_blocks = ["0.0.0.0/0"]
  }
}

resource "aws_key_pair" "ssh_key" {
  key_name = "ssh_key"
  public_key = file("ssh_key.pub")
}
```

This configuration uses the `file` function to read the contents of a file to configure an SSH key pair. The `file` function does not interpolate values into file contents; you should only use it with files that do not need modification.

Next, edit your `aws_instance.web` resource to use the new security group and key pair. Be sure to save your changes.

main.tf

```
resource "aws_instance" "web" {
  ami                         = data.aws_ami.ubuntu.id
  instance_type               = "t2.micro"
  subnet_id                   = aws_subnet.subnet_public.id
- vpc_security_group_ids      = [aws_security_group.sg_8080.id]
+ vpc_security_group_ids      = [aws_security_group.sg_22.id,
aws_security_group.sg_8080.id]
  associate_public_ip_address = true
  user_data                   = templatefile("user_data.tftpl", { department =
var.user_department, name = var.user_name })
+ key_name                    = aws_key_pair.ssh_key.key_name
}
```

Warning: This configuration enables public SSH traffic to the example instance for tutorial purposes. Lock down access to your services in production environments.

Apply your configuration to create the resources. Enter `yes` when prompted to confirm the operation.

```
$ terraform apply
##...
Plan: 9 to add, 0 to change, 0 to destroy.

Changes to Outputs:
  + ami_value          = "ami-0739f8cdb239fe9ae"
  + web_public_address = (known after apply)
  + web_public_ip      = (known after apply)

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

##...
```

To confirm that your instance now accepts traffic on port 22, SSH into it from your terminal.

Note: It may take up to 5 minutes to provision your instance. If you receive a `public key` error message, wait a couple minutes before trying again.

```
$ ssh ubuntu@$(terraform output -raw web_public_ip) -i ssh_key
```

Clean up resources

Now that you have completed this tutorial, destroy the resources to avoid incurring unnecessary costs. Respond `yes` when prompted to confirm the operation.

```
$ terraform destroy
##...
Plan: 0 to add, 0 to change, 9 to destroy.

Changes to Outputs:
  - ami_value          = "ami-0739f8cdb239fe9ae" -> null
  - web_public_address = "3.84.177.194:8080" -> null
  - web_public_ip      = "3.84.177.194" -> null

Do you really want to destroy all resources?
  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
##...
Apply complete! Resources: 0 added, 0 changed, 9 destroyed.
```

If you used Terraform Cloud for this tutorial, after destroying your resources, delete the `learn-terraform-functions` workspace from your Terraform Cloud organization.

Next steps

In this tutorial, you learned how to make your Terraform configuration dynamic by using built-in functions. You used the `lookup` function to access values from maps based on an input variable, the `templatefile` function to generate a script with interpolated values, and the `file` function to use the contents of a file as-is within configuration.

Check out the following resources to learn more about how to make your Terraform configuration more flexible:

- Review the functions documentation to learn more about the functions Terraform supports.
- Learn how to manage similar resources using `count`.
- Learn how to create dynamic expressions in your configuration.
- Use `for_each` to dynamically configure your resources based on a map.