terraform provisioner

Table of Content

Terraform Provisioners are used to performing certain custom actions and tasks either on the local machine or on the remote machine.

The custom actions can vary in nature and it can be -

1. Running custom shell script on the local machine
2. Running custom shell script on the remote machine
3. Copy file to the remote machine

Also, **there are two types of provisioners -**

1. Generic Provisioners (file, local-exec, and remote-exec)
2. Vendor Provisioners (chef, habitat, puppet, salt-masterless)

Generic Provisioners - Generally vendor independent and can be used with any cloud vendor(GCP, AWS, AZURE)

Vendor Provisioners - It can only be used only with its vendor. For example, `chef` provisioner can only be used with chef for automating and provisioning the server configuration.

1. file provisioner

As the name suggests *file provisioner* can be used for transferring and copying the files from one machine to another machine.

Not only file but it can also be used for transferring/uploading the directories.

So when we talk about copying files or directories from one machine to another machine then it has to be secured and *file provisioner* supports for `ssh` and `winrm` type of connections which can help you to achieve secure file transfer between the source machine and destination machine.

Let us take an example to understand how to implement terraform file provisioner. The following code snippet shows -

1. How to write your file provisioner
2. How to specify `source and  destination` for copying/transferring the file.

```
provisioner "file" {
    source      = "/home/rahul/Jhooq/keys/aws/test-file.txt"
    destination = "/home/ubuntu/test-file.txt"
}
```

In the above code snippet, we are trying to copy file `test-file.txt` from its **source =**`/home/rahul/Jhooq/keys/aws/test-file.txt` to its **destination =**`/home/ubuntu/test-file.txt`

Here is the complete terraform script which demonstrates on how to use *terraform file provisioner*

```
provider "aws" {
   region     = "eu-central-1"
   access_key = "AKIATQ37NXBxxxxxxxxx"
   secret_key = "JzZKiCia2vjbq4zGGGewdbOhnacmxxxxxxxxxxx"

}

resource "aws_instance" "ec2_example" {
```

```
    ami = "ami-0767046d1677be5a0"
    instance_type = "t2.micro"
    key_name= "aws_key"
    vpc_security_group_ids = [aws_security_group.main.id]

  provisioner "file" {
    source      = "/home/rahul/Jhooq/keys/aws/test-file.txt"
    destination = "/home/ubuntu/test-file.txt"
  }
  connection {
      type        = "ssh"
      host        = self.public_ip
      user        = "ubuntu"
      private_key = file("/home/rahul/Jhooq/keys/aws/aws_key")
      timeout     = "4m"
   }
}

resource "aws_security_group" "main" {
  egress = [
    {
      cidr_blocks      = [ "0.0.0.0/0", ]
      description      = ""
      from_port        = 0
      ipv6_cidr_blocks = []
      prefix_list_ids  = []
      protocol         = "-1"
      security_groups  = []
      self             = false
      to_port          = 0
    }
  ]
 ingress                = [
   {
     cidr_blocks      = [ "0.0.0.0/0", ]
     description      = ""
     from_port        = 22
     ipv6_cidr_blocks = []
     prefix_list_ids  = []
     protocol         = "tcp"
     security_groups  = []
     self             = false
     to_port          = 22
  }
  ]
}

resource "aws_key_pair" "deployer" {
  key_name   = "aws_key"
  public_key = "ssh-rsa AAAAB3NzaC1yc2EAAAADAQABAAABAQDbvRN
```

```
/gvQBhFe+dE8p3Q865T
/xTKgjqTjj56p1IIKbq8SDyOybE8ia0rMPcBLAKds+wjePIYpTtRxT9UsUbZJTgF+SGSG2dC
6+ohCQpi6F3xM7ryL9fy3BNCT5aPrwbR862jcOIfv7R1xVfH8OS0WZa8DpVy5kTeutsuH5FM
AmEgba4KhYLTzIdhM7UKJvNoUMRBaxAqIAThqH9Vt
/iR1WpXgazoPw6dyPssa7ye6tUPRipmPTZukfpxcPlsqytXWlXm7R89xAY9OXkdPPVsrQA0X
FQnY8aFb9XaZP8cm7EOVRdxMsA1DyWMVZOTjhBwCHfEIGoePAS3jFMqQjGWQd
rahul@rahul-HP-ZBook-15-G2"
}
```

Here is one thing to note - *You need to generate the ssh keys to connect to your EC2 instance running in the AWS cloud. You can use the command* `ssh-keygen -t aws_key` to generate the key-pair. You can read this blog post on Terraform how to do SSH in AWS EC2 instance?

Supporting arguments for file provisioners

1. source -

The source argument is used to specify the location from where you want to pick the file. The source location can be relative to your project structure.

Here are some examples where I have used relative path for the source arguments -

```
provisioner "file" {
    source      = "../../../Jhooq/keys/aws/test-file.txt"
    destination = "/home/ubuntu/test-file.txt"
}
```

1. content -

The content argument is useful when you do not want to copy or transfer the file instead you only want to copy the `content` or `string`.

Here is an example of a content resource argument -

```
provisioner "file" {
    content     = "I want to copy this string to the destination file
server.txt"
    destination = "/home/ubuntu/server.txt"
}
```

The above provisioner script will copy the string `I want to copy this string to the destination file server.txt` to the destination file `/home/ubuntu/server.txt`

1. destination -

As the name suggests you need to input the final destination path where you want your file to be.

2. local-exec provisioner

The next provisioner we are gonna talk about is the *local-exec provisioner*. Basically, this provisioner is used when you want to perform some tasks onto your local machine where you have installed the terraform.

So `local-exec` provisioner is never used to perform any kind of task on the remote machine. It will always be used to perform local operations onto your local machine.

Example - Consider the following example where we are trying to create a file `hello-jhooq.txt` on the local machine

```
provisioner "local-exec" {
    command = "touch hello-jhooq.txt"
}
```

In the command section, we can write a bash script. In the above example, I am trying to create a `hello-jhooq.txt` file on the local machine.

Here is the complete terraform script for the above example -

```
provider "aws" {
   region     = "eu-central-1"
   access_key = "AKIATQ37NXBxxxxxxxxx"
   secret_key = "JzZKiCia2vjbq4zGGGewdbOhnacmxxxxxxxxxxxx"

}

resource "aws_instance" "ec2_example" {

    ami = "ami-0767046d1677be5a0"
    instance_type = "t2.micro"
    tags = {
        Name = "Terraform EC2"
    }

  provisioner "local-exec" {
    command = "touch hello-jhooq.txt"
  }
}
```

Supporting arguments for local provisioners

1. command -

Here are the key facts about the ***command*** arguments

1. This is a mandatory argument that you always need to pass along whenever you are implementing the `local-exec` provisioners.
2. Always consider command as shell script executioner because whatever you pass in the command will be executed as a bash shell script.

3. You can write even mention the relative path of your shell script location and pass it the command.

```
provisioner "local-exec" {
    command = "touch hello-jhooq.txt"
}
```

1. working_dir -

Here are the key facts about the `working_dir` arguments

1. It is an optional argument and you do not necessarily need to pass along with the command argument.
2. This is a supporting argument for the `command` because once you specify the `working_dir` you are explicitly telling terraform to execute the command at that particular location.
3. You can mention the relative path of the `working_dir`.

1. interpreter -

With the help of an `interpreter` you can explicitly specify in which environment(bash, PowerShell, Perl, etc.) you are going to execute the command.

1. It is an optional argument.
2. If you do not specify the `interpreter` argument the default will be taken into consideration based on the operating system.

Example 1: - Here I am trying to specify the interpreter as Perl, so anything which I mention inside the command argument will be executed as Perl command

```
resource "null_resource" "example1" {
  provisioner "local-exec" {
    command = "open WFH, '>hello-world.txt' and print WFH scalar
localtime"
    interpreter = ["perl", "-e"]
  }
}
```

Example 2: - In this example, I will be using the `PowerShell` interpreter to write a string to a file

```
resource "null_resource" "example2" {
  provisioner "local-exec" {
    command = "This will be written to the text file> completed.txt"
    interpreter = ["PowerShell", "-Command"]
  }
}
```

BASH

1. environment -

This is again an optional parameter that can be passed alongside the `command` argument.

1. With the help of `environment` you can define or set environment variables that can be accessible later or inside your terraform execution.
2. `environment` arguments are generally the key-value pair and you can define as many variables as you can.

Here is an example of the `environment` arguments

```
provisioner "local-exec" {
    command = "echo $VAR1 $VAR2 $VAR3 >> my_vars.txt"

    environment = {
      VAR1 = "my-value-1"
      VAR2 = "my-value-2"
      VAR3 = "my-value-3"
    }
  }
```

BASH

3. remote-exec provisioner

As the name suggests `remote-exec` it is always going to work on the remote machine. With the help of the `remote-exec` you can specify the commands of shell scripts that want to execute on the remote machine.

As we discussed `ssh` and `winrm` for secure data transfer in local-exec, here also all the communication and file transfer is done securely.

Let us take an example of how to implement the `remote-exec` provisioner -

```
provisioner "remote-exec" {
    inline = [
      "touch hello.txt",
      "echo helloworld remote provisioner >> hello.txt",
    ]
}
```

BASH

In the above example -

1. First we are going to create a file named `hello.txt`

   We are going to write the message `helloworld remote provisioner` inside the `hello.txt` file.
2. Everything will happen on the remote machine

Here is the complete example of `remote-exec` -

```
provider "aws" {
   region     = "eu-central-1"
   access_key = "AKIATQ37NXBxxxxxxxxx"
   secret_key = "JzZKiCia2vjbq4zGGGewdbOhnacmxxxxxxxxxxxx"

}

resource "aws_instance" "ec2_example" {

    ami = "ami-0767046d1677be5a0"
```

```
    instance_type = "t2.micro"
    key_name= "aws_key"
    vpc_security_group_ids = [aws_security_group.main.id]

  provisioner "remote-exec" {
    inline = [
      "touch hello.txt",
      "echo helloworld remote provisioner >> hello.txt",
    ]
  }
  connection {
      type        = "ssh"
      host        = self.public_ip
      user        = "ubuntu"
      private_key = file("/home/rahul/Jhooq/keys/aws/aws_key")
      timeout     = "4m"
   }
}

resource "aws_security_group" "main" {
  egress = [
    {
      cidr_blocks      = [ "0.0.0.0/0", ]
      description      = ""
      from_port        = 0
      ipv6_cidr_blocks = []
      prefix_list_ids  = []
      protocol         = "-1"
      security_groups  = []
      self             = false
      to_port          = 0
    }
  ]
 ingress                = [
   {
     cidr_blocks      = [ "0.0.0.0/0", ]
     description      = ""
     from_port        = 22
     ipv6_cidr_blocks = []
     prefix_list_ids  = []
     protocol         = "tcp"
     security_groups  = []
     self             = false
     to_port          = 22
  }
  ]
}


resource "aws_key_pair" "deployer" {
```

```
  key_name   = "aws_key"
  public_key = "ssh-rsa AAAAB3NzaC1yc2EAAAADAQABAAABAQDbvRN
/gvQBhFe+dE8p3Q865T
/xTKgjqTjj56p1IIKbq8SDyOybE8ia0rMPcBLAKds+wjePIYpTtRxT9UsUbZJTgF+SGSG2dC
6+ohCQpi6F3xM7ryL9fy3BNCT5aPrwbR862jcOIfv7R1xVfH8OS0WZa8DpVy5kTeutsuH5FM
AmEgba4KhYLTzIdhM7UKJvNoUMRBaxAqIAThqH9Vt
/iR1WpXgazoPw6dyPssa7ye6tUPRipmPTZukfpxcPlsqytXWlXm7R89xAY9OXkdPPVsrQA0X
FQnY8aFb9XaZP8cm7EOVRdxMsA1DyWMVZOTjhBwCHfEIGoePAS3jFMqQjGWQd
rahul@rahul-HP-ZBook-15-G2"
}
```

...

BASH

Supporting arguments for remote provisioners

1. inline -

With the help of an inline argument you can specify the multiple commands which you want to execute in an ordered fashion.

Here is an example in which I have added two separate commands -

```
provisioner "remote-exec" {
    inline = [
      "touch hello.txt",
      "echo helloworld remote provisioner >> hello.txt",
    ]
}
```

BASH

1. script -

It can be used to copy the script from local machine to remote machine and it always contains a relative path.

In the script, you can not specify multiple scripts. You can only mention one script which needs to be copied to the remote machine.

1. scripts -

Here you can specify the multiple local scripts which want to copy or transfer to the remote machine and execute over there.

Always remember the order of the file will not change and it going to execute in the same order way you have mentioned.