**Provisioners**

**Define a Provisioner**

We're now able to create and modify infrastructure. Now let's see how to use provisioners to initialize instances when they're created. If you need to do some initial setup on your instances, provisioners let you upload files, run shell scripts, or install and trigger other software like configuration management tools, etc.

To define a **provisioner**, modify the resource block defining the "example" EC2 instance to look like the following:

resource "aws\_instance" "example" {

ami = "ami-b374d5a5"

instance\_type = "t2.micro"

provisioner "local-exec" {

command = "echo ${aws\_instance.example.public\_ip} > ip\_address.txt"

}

}

This adds a provisioner block within the resource block. Multiple provisioner blocks can be added to define multiple provisioning steps. Terraform supports multiple provisioners, but for this example we are using the local-exec provisioner.

Run terraform init and terraform apply and observe the local-exec provisioner executing a command locally on your machine running Terraform. We're using this provisioner versus the others so we don't have to worry about specifying any connection info right now. The local-exec provisioner you just ran created a file called ip\_address.txt on your local machine where you ran your terraform apply command.

$ cat ip\_address.txt

54.89.98.96

Another useful provisioner is remote-exec which invokes a script on a remote resource after it is created. **This can be used to run a configuration management tool**, bootstrap into a cluster, etc. In order to use a remote-exec provisioner, you must choose an **ssh** or **winrm** connection in the form of a connection block within the provisioner. Here is an example of how to use remote-exec to *install a specific package* on a single instance at startup. You should have an ssh key created with appropriate permissions to run the example below.

Create an ssh key with no passphrase with ssh-keygen -t rsa and use the name terraform. Update the permissions of that key with chmod 400 ~/.ssh/terraform.

This example is for reference and should not be used without testing. If you are running this, create a new Terraform project folder for this example.

provider "aws" {

profile = "default"

region = "us-west-2"

}

resource "aws\_key\_pair" "example" {

key\_name = "examplekey"

public\_key = file("~/.ssh/terraform.pub")

}

resource "aws\_instance" "example" {

key\_name = aws\_key\_pair.example.key\_name

ami = "ami-04590e7389a6e577c"

instance\_type = "t2.micro"

connection {

type = "ssh"

user = "ec2-user"

private\_key = file("~/.ssh/terraform")

host = self.public\_ip

}

provisioner "remote-exec" {

inline = [

"sudo amazon-linux-extras enable nginx1.12",

"sudo yum -y install nginx",

"sudo systemctl start nginx"

]

}

}

This example has a few pieces to go over. The initial resource for the aws\_key\_pair is required for **SSH** connections. You must create a keypair locally to upload to AWS and the aws\_key\_pair resource is the function for that. The aws\_instance resource needs the key\_name connected to it directly as an attribute. Within the aws\_instance resource, we create a connection block which must define the connection type, the user, host, and private\_key attributes.

The private\_key attribute is necessary to successfully provision the host. Once that connection is successful, the remote-exec provisioner will run on the remote host to **install, update, and start nginx** in this example.

### Running Provisioners

Provisioners are **only run when a resource is created**. They are not a replacement for configuration management and changing the software of an already-running server, and are instead just meant as a way to bootstrap a server.

Make sure that your infrastructure is destroyed if it isn't already, then run apply:

$ terraform apply

# ...

aws\_instance.example: Creating...

ami: "" => "ami-b374d5a5"

instance\_type: "" => "t2.micro"

aws\_eip.ip: Creating...

instance: "" => "i-213f350a"

Apply complete! Resources: 2 added, 0 changed, 0 destroyed.

Terraform will output anything from provisioners to the console, but in this case there is no output. However, we can verify everything worked by looking at the ip\_address.txt file:

$ cat ip\_address.txt

54.192.26.128

It contains the IP, just as we asked!

### Failed Provisioners and Tainted Resources

If a resource successfully creates but fails during provisioning, Terraform will error and mark the resource as **"tainted"**. A resource that is tainted has been physically created, but can't be considered safe to use since provisioning failed.

When you generate your next execution plan, Terraform will not attempt to restart provisioning on the same resource because it isn't guaranteed to be safe. Instead, Terraform will remove any tainted resources and create new resources, attempting to provision them again after creation.

Terraform also does not automatically roll back and destroy the resource during the apply when the failure happens, because that would go against the execution plan: the execution plan would've said a resource will be created, but does not say it will ever be deleted. If you create an execution plan with a tainted resource, however, the plan will clearly state that the resource will be destroyed because it is tainted.

### Manually Tainting Resources

In cases where you want to manually destroy and recreate a resource, Terraform has a built in taint function in the CLI. This command will not modify infrastructure, but does modify the state file in order to mark a resource as tainted. Once a resource is marked as tainted, the next plan will show that the resource will be destroyed and recreated and the next apply will implement this change.

To taint a resource, use the following command:

terraform taint resource.id

resource.id refers to the resource block name and resource ID to taint. Review the resource block we previously created:

resource "aws\_instance" "example" {

ami = "ami-b374d5a5"

instance\_type = "t2.micro"

}

The correct resource and ID to taint this resource would be terraform taint aws\_instance.example.

### Destroy Provisioners

⚜ Provisioners can also be defined that run only during a destroy operation. These are useful for performing system cleanup, extracting data, etc.

⚜ For many resources, using built-in cleanup mechanisms is recommended if possible (such as init scripts), but provisioners can be used if necessary.

We won't show any destroy provisioner examples. If you need to use destroy provisioners, please see [the provisioner documentation](https://www.terraform.io/docs/provisioners/" \t "_blank) .