Terraform Provisioners

Terraform Provisioners are used for executing scripts or shell commands on a local or remote machine as part of resource creation/deletion. They are similar to "**AWS-EC2 instance-user data**" scripts that only run once on the creation and if it fails terraform marks it tainted.

Why are provisioners used as a last resort?

Terraform includes the concept of provisioners as a measure of pragmatism, knowing that there will always be certain behaviors that can't be directly represented in Terraform's declarative model.

Provisioners can be used to model specific actions on the local machine or on a remote machine in order to prepare servers or other infrastructure objects for service.

Terraform includes the concept of provisioners as a measure of pragmatism, knowing that there will always be certain behaviors that can't be directly represented in Terraform's declarative model.

However, they also add considerable complexity and uncertainty to Terraform usage. Firstly, Terraform cannot model the actions of provisioners as part of a plan because they can in principle take any action. Secondly, successful use of provisioners requires coordinating more details than Terraform usage usually requires: direct network access to your servers, issuing Terraform credentials to log in, making sure that all of the necessary external software is installed, etc.

How to use provisioners:

If you are certain that provisioners are the best way to solve your problem, you can add a provisioner block inside the resource block of a compute instance.

```
resource "aws_instance" "web" {
  # ...

  provisioner "local-exec" {
    command = "echo The server's IP address is
${self.private_ip}"
  }
}
```

The local-exec provisioner requires no other configuration, but most other provisioners must connect to the remote system using SSH or WinRM. You must include a connection block so that Terraform will know how to communicate with the server.

Creation-time provisioners
By default, provisioners run when the resource they are defined within is created. Creation-time provisioners are only run during creation, not during updating or any other lifecycle. They are meant as a means to perform bootstrapping of a system. So, if a creation-time provisioner fails, the resource is marked as tainted. A tainted resource will be planned for destruction and recreation upon the next terraform apply.

Destroy-time provisioners
If ***when = destroy*** is specified, the provisioner will run when the resource it is defined within is destroyed.
Destroy provisioners are run before the resource is destroyed. If they fail, Terraform will error and rerun the provisioners again on the next terraform apply. Due to this behavior, care should be taken to destroy provisioners to be safe to run multiple times.

```
resource "aws_instance" "web" {
  # ...

  provisioner "local-exec" {
    when    = destroy
    command = "echo 'Destroy-time provisioner'"
  }
}
```

Multiple provisioners
Multiple provisioners can be specified within a resource. Multiple provisioners are executed in the order they’re defined in the configuration file.
You may also mix and match creation and destruction provisioners. Only the provisioners that are valid for a given operation will be run. Those valid provisioners will be run in the order they’re defined in the configuration file.

```
resource "aws_instance" "web" {
  # ...

  provisioner "local-exec" {
    command = "echo first"
  }

  provisioner "local-exec" {
    command = "echo second"
```

```
  }
}
```

Failure Behavior

By default, provisioners that fail will also cause the Terraform apply itself to fail. The ***on_failure*** setting can be used to change this. The allowed values are:

- continue – Ignore the error and continue with creation or destruction.
- fail – Raise an error and stop applying (the default behavior). If this is a creation provisioner, taint the resource.

```
resource "aws_instance" "web" {
  # ...

  provisioner "local-exec" {
    command    = "echo The server's IP address is
${self.private_ip}"
    on_failure = continue
  }
}
```

Types of Provisioners in terraform

There are two types of provisioners:

1. **Generic Provisioners:** These are generally vendor independent and can be used with any cloud vendor.
   Ex: file, local-exec, and remote-exec.
2. **Vendor Provisioners:** It can only be used only with its vendor.
   Ex: Chef provisioner can only be used with the chef for automating and provisioning the server configuration, habitat, puppet, and salt-masterless

Let's look at each of these provisions in brief with code examples.

File Provisioner:

The file provisioner can upload a complete directory to the remote machine.

```
resource "aws_instance" "web" {
  # ...

  # Copies the myapp.conf file to /etc/myapp.conf
  provisioner "file" {
    source      = "conf/myapp.conf"
    destination = "/etc/myapp.conf"
  }

  # Copies the string in content into /tmp/file.log
  provisioner "file" {
    content     = "ami used: ${self.ami}"
```

```
    destination = "/tmp/file.log"
  }

  # Copies all files and folders in apps/app1 to
D:/IIS/webapp1
  provisioner "file" {
    source      = "apps/app1/"
    destination = "D:/IIS/webapp1"
  }
}
```

local-exec Provisioner:
The local-exec provisioner invokes a local executable after a resource is created. This invokes a process on the machine running Terraform, not on the resource. Basically, this provisioner is used when you want to perform some tasks onto your local machine where you have installed the terraform. So the local-exec provisioner is never used to perform any task on the remote machine. It will always be used to perform local operations onto your local machine.

```
resource "aws_instance" "web" {
  # ...

  provisioner "local-exec" {
    command = "echo ${self.private_ip} >> private_ips.txt"
  }
}
```

remote-exec Provisioner:
The remote-exec provisioner 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. It requires a connection and supports both ssh and winrm.

```
resource "aws_instance" "web" {
  # ...

  # Establishes connection to be used by all
  # generic remote provisioners (i.e. file/remote-exec)
  connection {
    type     = "ssh"
    user     = "root"
    password = var.root_password
    host     = self.public_ip
  }
```

```
  provisioner "remote-exec" {
    inline = [
      "puppet apply",
      "consul join ${aws_instance.web.private_ip}",
    ]
  }
}
```

It can be used inside the Terraform resource object and in that case, it will be invoked once the resource is created, or it can be used inside a null resource which is my preferred approach as it separates this non-terraform behavior from the real terraform behavior.

Chef Provisioner:
Chef is an infrastructure-automating open-source tool for managing and automating the servers remotely. Terraform has really good integration with Chef. It installs, configures, and runs the Chef Client on a remote resource. The chef provisioner supports both ssh and Winrm type connections.

Habitat Provisioner:
The habitat provisioner installs the Habitat supervisor and loads configured services. This provisioner only supports Linux targets using the SSH connection type at this time. (This provisioner was removed in the 0.15.0 version of Terraform after being deprecated as of Terraform 0.13.4.)

Puppet Provisioner:
The Puppet provisioner installs, configures, and runs the Puppet agent on a remote resource. The puppet provisioner supports both SSH and Winrm type connections.

Salt Masterless Provisioner:
The salt-masterless Terraform provisioner provisions machines built by Terraform using Salt states, without connecting to a Salt master. The salt-masterless provisioner supports SSH connections. The salt-masterless provisioner has some prerequisites. cURL must be available on the remote host.
So this was all about Terraform Provisioners, their types, and usage.

Conclusion:
Terraform provisioners play a role in specific scenarios, but their use should be approached with caution. Strive to maintain the declarative nature of Terraform whenever possible and leverage provisioners only when absolutely necessary. By understanding the limitations and trade-offs, you can make informed decisions and create more maintainable and scalable infrastructure code.

FAQ

1. **When should I use Terraform provisioners?**
   Provisioners should be considered only when there's no other option to achieve a specific configuration using Terraform's built-in resources and modules. Use them sparingly for tasks like bootstrapping or configuration that cannot be expressed declaratively.

2. **Are there alternatives to Terraform provisioners?**
   Yes, consider using Terraform's built-in resources, modules, and external tools (like Ansible, Chef, or Puppet) before resorting to provisioners. Leverage the power of declarative syntax as much as possible.

3. **How can I maintain idempotence when using provisioners?**
   Carefully design and test your provisioner scripts to ensure they maintain idempotence. Always aim for scripts that produce the same result when executed multiple times.

4. **What are the common pitfalls when using Terraform provisioners?**
   Common pitfalls include introducing non-declarative logic into your infrastructure, increased complexity, and potential dependency issues with external tools or scripts. Always weigh the pros and cons before deciding to use provisioners.