Terraform provisioner:

Terraform provisioners are used to execute scripts or commands on a local or remote machine as part of the resource creation process. They allow you to perform configuration tasks that cannot be done through Terraform’s built-in resource types.

**Types of Provisioners**

1. **Local Exec Provisioner**: Executes a command on the machine running Terraform.
2. **Remote Exec Provisioner**: Executes a command on the resource being created or managed.

**Local Exec Provisioner**

The local-exec provisioner runs commands on the machine where terraform apply is executed.

**Example**

resource "aws\_instance" "example" {

ami = "ami-123456"

instance\_type = "t2.micro"

provisioner "local-exec" {

command = "echo ${aws\_instance.example.private\_ip} >> private\_ips.txt"

}

}

In this example, the local-exec provisioner appends the private IP of the newly created instance to a file called private\_ips.txt.

**Remote Exec Provisioner**

The remote-exec provisioner connects to the resource and runs commands remotely. This is typically used with virtual machines.

**Example**

resource "aws\_instance" "example" {

ami = "ami-123456"

instance\_type = "t2.micro"

provisioner "remote-exec" {

inline = [

"sudo apt-get update",

"sudo apt-get install -y nginx"

]

connection {

type = "ssh"

user = "ubuntu"

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

host = self.public\_ip

}

}

}

In this example, the remote-exec provisioner updates the package list and installs Nginx on an AWS EC2 instance.

**File Provisioner**

The file provisioner is used to copy files or directories from the machine running Terraform to the target resource.

**Example**

hcl

resource "aws\_instance" "example" {

ami = "ami-123456"

instance\_type = "t2.micro"

provisioner "file" {

source = "app.conf"

destination = "/etc/app.conf"

connection {

type = "ssh"

user = "ubuntu"

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

host = self.public\_ip

}

}

}

In this example, the file provisioner copies app.conf from the local machine to /etc/app.conf on the remote instance.

**Error Handling and Dependencies**

Provisioners can have error handling configurations to determine the behavior on failure:

* on\_failure = "continue": Ignores errors and continues with the remaining resources.
* on\_failure = "fail": Stops immediately if there is an error (default behavior).

**Example with Error Handling**

resource "aws\_instance" "example" {

ami = "ami-123456"

instance\_type = "t2.micro"

provisioner "remote-exec" {

inline = [

"sudo apt-get update",

"sudo apt-get install -y nginx"

]

connection {

type = "ssh"

user = "ubuntu"

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

host = self.public\_ip

}

on\_failure = "continue"

}

}

**Multiple Provisioners:** You can use multiple provisioners in a single resource. They will be executed in the order they are defined.

**Example**

resource "aws\_instance" "example" {

ami = "ami-123456"

instance\_type = "t2.micro"

provisioner "file" {

source = "app.conf"

destination = "/etc/app.conf"

}

provisioner "remote-exec" {

inline = [

"sudo systemctl restart nginx"

]

}

}

In this example, the file is copied first, and then Nginx is restarted.

**Best Practices**

* **Minimize Provisioner Use**: Use Terraform's built-in resources for configuration as much as possible.
* **Idempotency**: Ensure commands are idempotent to avoid unintended consequences on re-apply.
* **Error Handling**: Implement proper error handling to manage failures gracefully.
* **Use Configuration Management Tools**: For complex provisioning, consider using dedicated configuration management tools like Ansible, Chef, or Puppet instead of inline provisioners.

Provisioners are powerful, but they should be used judiciously and primarily as a last resort when other Terraform-native options are insufficient.