Terraform Provisioners: Real-World DevOps in Action with Interview Ques!

Connect with me: Amit Singh

What are Terraform Provisioners?

Terraform Provisioners are used to **execute scripts or commands** on a local or remote machine **after a resource is created** or **before it's destroyed**. They act as a bridge between Terraform's infrastructure as code and configuration management.

Types of Provisioners:

Type	Description
file	**Used to copy files from local machine to remote resource (e.g., EC2).**
remote-exec	**Executes shell commands on the remote machine (e.g., install packages).**
local-exec	**Executes shell commands on the machine where Terraform is run.**

Provisioner Use Cases in DevOps

- Bootstrapping servers (installing packages, apps, or services)
- Running initialization scripts
- Performing cleanups during terraform destroy
- Avoiding manual SSH and repetitive scripts
- Integrating with CI/CD tools (Jenkins, GitHub Actions)

⚠ **Important:** Provisioners should be used **as a last resort**. HashiCorp recommends avoiding provisioners when possible in favor of external tooling or more robust IaC patterns.

Default Behavior of Provisioners

- By default, they execute **after resource creation**.
- To run on **destroy**, you need to specify when = "destroy" inside the provisioner block.

```
provisioner "remote-exec" {
```

```
  when    = "destroy"

  inline  = ["echo Destroying instance..."]

}
```

Why Not Use Provisioners Always?

Although powerful, provisioners:

- Break **idempotency** of Terraform
- Are considered a **last resort** by HashiCorp (if other methods like AMIs or cloud-init don't work)
- Can lead to **errors if connection fails** or scripts are unstable

Best Practice: Use provisioners *only when necessary*, and prefer using tools like Ansible, Packer, or baked AMIs for production-grade provisioning.

Common Pitfalls (Observed)

- Flask app didn't persist as a background process → Use nohup or & in the command:

```
inline = [

  "nohup python3 /home/ubuntu/app.py &"

      ]
```

- Ensure the security group allows **port 5000** (Flask default) for public access.
- Use user_data as an alternative for one-time EC2 setup tasks (but less flexible than provisioners).

DevOps Wisdom from Provisioners

A DevOps Engineer isn't just writing Terraform. They are **enabling developers** by eliminating repetitive tasks, automating infrastructure + config setup, and ensuring delivery with zero manual intervention.

This is where **Dev + Ops** meet: Developers get what they need with **just a Terraform apply** or a **Jenkins job trigger**.

Dev Team Request: Automate their Testing Environment **Real-World Scenario (Practical First)**

Let’s assume a dev team (Team XYZ) has the following request:

“Every time we update our app.py file, we want an EC2 instance to be provisioned with the latest code deployed, accessible over the Internet.”

🗒️ Requirements

- VPC creation
- Public Subnet
- Internet Gateway
- Route Table for public routing
- EC2 instance with the latest app.py Flask code
- Security Group with ports **22 (SSH)** and **80 (HTTP)** open
- Access the app via http://<public-ip>:80

Why Not Use EC2 User Data?

While **user data** is useful for quick provisioning scripts, it becomes **unmanageable for complex applications** with:

- Multiple dependencies
- Dozens of files or folders
- Git-based or external code deployment

That’s where **Terraform Provisioners** come in handy:

- **file provisioner:** Upload files to EC2
- **remote-exec:** Run commands on the EC2 instance

Step-by-Step Terraform Project Structure

✅ Step 1: Define AWS Provider (main.tf)

```
# Define the AWS provider configuration.
provider "aws" {
  region = "us-east-1"  # Replace with your desired AWS region.
}
```

✅ Step 2: Create Key Pair

Generate it using:

$ssh-keygen -t rsa

Then use the public key in your Terraform configuration.

```
variable "cidr" {
  default = "10.0.0.0/16"
}

resource "aws_key_pair" "example" {
  key_name   = "terraform-demo-abhi"  # Replace with your desired key name
  public_key = file("~/.ssh/id_rsa.pub")  # Replace with the path to your public key file
}
```

✅ Step 3: Create VPC

```
resource "aws_vpc" "myvpc" {
  cidr_block = var.cidr
}
```

✅ Step 4: Create Subnet

```
resource "aws_subnet" "sub1" {
  vpc_id                  = aws_vpc.myvpc.id
  cidr_block              = "10.0.0.0/24"
  availability_zone       = "us-east-1a"
  map_public_ip_on_launch = true
}
```

✅ Step 5: Internet Gateway & Route Table

```
resource "aws_internet_gateway" "igw" {
  vpc_id = aws_vpc.myvpc.id
}

resource "aws_route_table" "RT" {
  vpc_id = aws_vpc.myvpc.id

  route {
    cidr_block = "0.0.0.0/0"
    gateway_id = aws_internet_gateway.igw.id
  }
}

resource "aws_route_table_association" "rta1" {
  subnet_id      = aws_subnet.sub1.id
  route_table_id = aws_route_table.RT.id
}
```

✅ Step 6: Create Security Group

```
resource "aws_security_group" "webSg" {
  name   = "web"
  vpc_id = aws_vpc.myvpc.id

  ingress {
    description = "HTTP from VPC"
    from_port   = 80
    to_port     = 80
    protocol    = "tcp"
    cidr_blocks = ["0.0.0.0/0"]
  }
  ingress {
    description = "SSH"
    from_port   = 22
    to_port     = 22
    protocol    = "tcp"
    cidr_blocks = ["0.0.0.0/0"]
  }
```

```
  egress {
    from_port   = 0
    to_port     = 0
    protocol    = "-1"
    cidr_blocks = ["0.0.0.0/0"]
  }

  tags = {
    Name = "Web-sg"
  }
}
```

✅ Step 7: Launch EC2

```
resource "aws_instance" "server" {
  ami                    = "ami-0261755bbcb8c4a84"
  instance_type          = "t2.micro"
  key_name      = aws_key_pair.example.key_name
  vpc_security_group_ids = [aws_security_group.webSg.id]
  subnet_id              = aws_subnet.sub1.id

  connection {
    type        = "ssh"
    user        = "ubuntu"  # Replace with the appropriate username for your EC2 instance
    private_key = file("~/.ssh/id_rsa")  # Replace with the path to your private key
    host        = self.public_ip
  }
```

✅ Step 8:Use Provisioners

```
  # File provisioner to copy a file from local to the remote EC2 instance
  provisioner "file" {
    source      = "app.py"  # Replace with the path to your local file
    destination = "/home/ubuntu/app.py"  # Replace with the path on the remote instance
  }

  provisioner "remote-exec" {
    inline = [
      "echo 'Hello from the remote instance'",
      "sudo apt update -y",  # Update package lists (for ubuntu)
      "sudo apt-get install -y python3-pip",  # Example package installation
      "cd /home/ubuntu",
      "sudo pip3 install flask",
      "sudo python3 app.py &",
    ]
  }
}
```

This entire setup will be automated using **Terraform**, and we used **Provisioners** to deploy the **app.py** file.

App.py

```
from flask import Flask

app = Flask(__name__)

@app.route("/")
def hello():
    return "Hello, Terraform!"

if __name__ == "__main__":
    app.run(host="0.0.0.0", port=80)
```

```
  from flask import Flask

  app = Flask(__name__)

  @app.route("/")
  def hello():
      return "Hello, Terraform!"

  if __name__ == "__main__":
      app.run(host="0.0.0.0", port=80)
```

Benefits of This Setup

- Fully automated VPC + EC2 + App deployment
- Devs don’t need to wait or create infrastructure manually

- Easily extensible for future use (e.g., Jenkins, GitHub Actions)

Pro Tip: Know the Manual First

If you **can't do it manually** in the AWS console, don't jump into Terraform yet.

Terraform is simply Infrastructure as Code — you need to **understand what you're codifying** first.

Let's delve deeper into the file, remote-exec, and local-exec provisioners in Terraform, along with examples for each.

1.file Provisioner:

The file provisioner is used to copy files or directories from the local machine to a remote machine. This is useful for deploying configuration files, scripts, or other assets to a provisioned instance.

Example:

```
resource "aws_instance" "example" {
  ami           = "ami-0c55b159cbfafe1f0"
  instance_type = "t2.micro"
}

provisioner "file" {
  source      = "local/path/to/localfile.txt"
  destination = "/path/on/remote/instance/file.txt"
  connection {
    type     = "ssh"
    user     = "ec2-user"
    private_key = file("~/.ssh/id_rsa")
  }
}
```

In this example, the file provisioner copies the localfile.txt from the local machine to the /path/on/remote/instance/file.txt location on the AWS EC2 instance using an SSH connection.

2.Remote-exec Provisioner:

The remote-exec provisioner is used to run scripts or commands on a remote machine over SSH or WinRM connections. It's often used to configure or install software on provisioned instances.

```
resource "aws_instance" "example" {
  ami           = "ami-0c55b159cbfafe1f0"
  instance_type = "t2.micro"
}

provisioner "remote-exec" {
  inline = [
    "sudo yum update -y",
    "sudo yum install -y httpd",
    "sudo systemctl start httpd",
  ]

  connection {
    type        = "ssh"
    user        = "ec2-user"
    private_key = file("~/.ssh/id_rsa")
    host        = aws_instance.example.public_ip
  }
}
```

In this example, the remote-exec provisioner connects to the AWS EC2 instance using SSH and runs a series of commands to update the package repositories, install Apache HTTP Server, and start the HTTP server.

3.local-exec Provisioner:
The local-exec provisioner is used to run scripts or commands locally on the machine where Terraform is executed. It is useful for tasks that don't require remote execution, such as initializing a local database or configuring local resources.

Example:

```
resource "null_resource" "example" {
  triggers = {
    always_run = "${timestamp()}"
  }

  provisioner "local-exec" {
    command = "echo 'This is a local command'"
  }
}
```

In this example, a null_resource is used with a local-exec provisioner to run a simple local command that echoes a message to the console whenever Terraform is applied or refreshed. The timestamp() function ensures it runs each time.

Top Terraform Provisioners Interview Questions

1. **What are provisioners in Terraform? When would you use them?**

- *Provisioners are used to execute scripts or commands on a local or remote machine after a resource is created. They're often used for bootstrapping instances or configuring servers temporarily when no config management tool is in place.*

2. **What types of provisioners are available in Terraform?**

- local-exec: Runs commands on the machine where Terraform is executed
- remote-exec: Runs commands on a remote resource (like an EC2 instance) over SSH or WinRM
- file: Uploads files from the local machine to the remote resource

3. **What are some risks or drawbacks of using provisioners?**

- Not idempotent (can cause inconsistent results)
- Error-prone in larger infrastructures
- May cause issues during resource destroy
- Terraform recommends avoiding them when possible and using tools like Ansible/Chef instead

4. **How do you pass SSH credentials to a remote-exec provisioner?**

- Using connection block:

```
connection {

 type       = "ssh"

 user       = "ec2-user"

 private_key = file("~/.ssh/key.pem")

 host       = self.public_ip

}
```

5. **What happens if a provisioner fails? How can you handle failures gracefully?**

- By default, Terraform fails the resource creation.
- You can use on_failure = continue to skip errors, but it should be used carefully.

6. **Can you explain how you used provisioners in a real project/demo?**

(Here's your answer for this exact demo)

In a Flask app demo, I used a file provisioner to copy app.py to an EC2 instance and a remote-exec provisioner to install Python and Flask, then run the app. This was helpful to bootstrap a minimal demo without setting up full config management tools like Ansible.

7. **When should you avoid using provisioners?**

- In production environments where configuration should be handled by CI/CD or dedicated tools like Ansible, Puppet, Chef, etc.
- When provisioning logic can be better handled with AMIs or cloud-init scripts

8. **What's the difference between provisioners and user-data in AWS EC2?**

- *User data* is part of EC2 instance metadata and runs during the first boot — good for simple scripts
- *Provisioners* run after Terraform creates the resource — more flexible but less reliable and not always idempotent

9. **How do you upload multiple files using the file provisioner?**

- Not directly supported — you'd either zip them or use remote-exec with a curl/wget/scp command to download from a remote repo

10. **How can you make provisioners conditional?**

```
provisioner "remote-exec" {

 when    = var.enable_provisioning ? "create" : "never"

 inline  = ["echo Provisioning only when enabled"]

}
```

🔹 Flashcard to remember quickly

Q: What are Terraform provisioners used for?
A: To execute scripts or commands after a resource is created, often for bootstrapping or configuration.

Q: Name 3 types of provisioners in Terraform.
A: local-exec, remote-exec, file

Q: What does the file provisioner do?
A: Uploads files from the local system to a remote resource.

Q: When should you avoid using provisioners?
A: In production or large-scale infra. Use config management tools instead (e.g., Ansible).

Q: What is remote-exec used for?
A: Executes commands on a remote server (like EC2) using SSH or WinRM.

Q: What is local-exec used for?
A: Runs a command **on the local machine** where Terraform is executed.

Q: How do you provide SSH credentials for remote provisioners?
A: Using the connection block with user, private_key, and host.

Q: How to handle a failed provisioner gracefully?
A: Use on_failure = "continue" (not recommended unless necessary).

Q: What's the difference between provisioner and user_data?
A: user_data runs at instance boot, provisioner runs after resource creation in Terraform.

Q: Can provisioners be conditional in Terraform?
A: Yes, using when and conditional logic with variables.

Thanks Everyone!

Connect with me: Amit Singh