

EC2 Provisioning with Terraform



solution to scenario -> Scenario – We want to execute some shell commands let's say installing nginx and starting the service on an ec2 instance. Ec2 will be created as part of terraform script and we don't want to touch(ssh) the ec2 instance. When we run terraform apply, our terraform file copy user-data.sh to ec2 using file provisioner and should install nginx and start nginx using remote_exec, depends_on ec2 is created first

But the issue is, if we change any command in user-data.sh and run terraform apply again then it will recreate the ec2 instance. Which is not the recommended way.

To avoid this we can maintain provisioners in a separate file let's say provisioners.tf

This will solve recreation issue but in this case, we need to restart ec2 instance to reflect changes done in user-data.sh

To resolve this issue we can use null_resource and filemd5() to detect changes.

To solve this issue efficiently, we can use Terraform's `null_resource` along with `filemd5()` to detect changes in `user-data.sh` without recreating the EC2 instance. Here's the refined approach:

Solution: Using `null_resource` and `filemd5()`

1. **Create an EC2 instance using Terraform** (without provisioners inside the EC2 resource).
2. **Use `null_resource` with `filemd5()`** to detect changes in `user-data.sh`.
3. **Use `remote-exec` to apply changes** only when `user-data.sh` is modified.

Terraform Code Structure

1. `main.tf` (EC2 Instance)

```
hcl

resource "aws_instance" "web" {
  ami           = "ami-0abcdef1234567890" # Replace with your AMI ID
  instance_type = "t2.micro"
  key_name      = "my-key" # Ensure you have an SSH key

  security_groups = ["default"]

  tags = {
    Name = "nginx-server"
  }
}
```

2. `provisioners.tf` (Provisioners using `null_resource`)

```
hcl

resource "null_resource" "install_nginx" {
  triggers = {
    script_hash = filemd5("${path.module}/user-data.sh") # Detect changes
  }
}
```

```

}

connection {
  type      = "ssh"
  user      = "ec2-user"
  private_key = file("~/ssh/my-key.pem") # Path to your private key
  host      = aws_instance.web.public_ip
}

provisioner "file" {
  source      = "user-data.sh"
  destination = "/tmp/user-data.sh"
}

provisioner "remote-exec" {
  inline = [
    "chmod +x /tmp/user-data.sh",
    "sudo /tmp/user-data.sh"
  ]
}

depends_on = [aws_instance.web]
}

```

3. `user-data.sh` (Script to Install & Start Nginx)

```

bash

#!/bin/bash
sudo yum update -y
sudo yum install -y nginx
sudo systemctl start nginx
sudo systemctl enable nginx

```

How This Works

- The `aws_instance.web` resource creates the EC2 instance **without running any provisioners**.
- The `null_resource.install_nginx` resource:
 - **Monitors changes** in `user-data.sh` using `filemd5()`.
 - **Triggers only when** `user-data.sh` **is modified**, preventing unnecessary EC2 recreation.
 - **Transfers** `user-data.sh` to the instance.
 - **Executes the script remotely** using `remote-exec`.

Advantages

- ✓ **Prevents EC2 recreation** when `user-data.sh` changes.
- ✓ **Automatically applies updates** to `user-data.sh` without manual intervention.
- ✓ **Keeps infrastructure modular** by separating EC2 creation and provisioning logic.

This approach ensures **Terraform apply only updates the script execution when needed**, avoiding full instance recreation. 🚀



explain local_exec in terraform

`local-exec` Provisioner in Terraform

The ``local-exec`` provisioner in Terraform is used to **execute commands on the local machine** where Terraform is running. Unlike ``remote-exec``, which runs commands on a remote machine (like an EC2 instance), ``local-exec`` executes commands on the **Terraform runner's system** (your local machine or CI/CD pipeline).

Syntax & Usage

Basic Example

```
hcl

resource "null_resource" "example" {
  provisioner "local-exec" {
    command = "echo 'Hello, Terraform!' > output.txt"
  }
}
```

✓ What Happens?

- This runs the ``echo`` command on your local machine.
- It creates an ``output.txt`` file with the text ``"Hello, Terraform!"``.

Use Cases of `local-exec`

1. Running Scripts Locally

You can use ``local-exec`` to trigger a local script after provisioning:

```
hcl

resource "null_resource" "run_script" {
  provisioner "local-exec" {
    command = "./deploy.sh"
  }
}
```

 **Example:** Running a deployment script from your local machine.

2. Logging Outputs

You can capture Terraform outputs using ``local-exec``:

```
hcl

resource "aws_instance" "example" {
  ami           = "ami-0abcdef1234567890"
  instance_type = "t2.micro"

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

```
}
}
```

✓ What Happens?

- After the instance is created, it appends the **public IP** to `instance_ips.txt`.

3. Sending Notifications

You can trigger notifications (Slack, Email, etc.) using `local-exec`:

```
hcl

resource "null_resource" "notify" {
  provisioner "local-exec" {
    command = "curl -X POST -H 'Content-type: application/json' --data '{"text": "Terraform Apply"
  }
}
```


 **Example:** Sends a Slack notification after Terraform completes.

4. Running Terraform Apply from a Parent Script

You can trigger another Terraform process:

```
hcl

resource "null_resource" "terraform_apply" {
  provisioner "local-exec" {
    command = "terraform apply -auto-approve"
  }
}
```

 **Use Case:** Automating Terraform workflow.

Handling Failures & `when` Argument

By default, `local-exec` runs during **resource creation**. You can control when it runs using `when`:

```
hcl

resource "aws_instance" "example" {
  provisioner "local-exec" {
    command = "echo 'Instance Created!'"
    when    = destroy
  }
}
```

✓ **Runs only during destroy phase.**

Key Takeaways

- ✓ **Runs on the local machine (not remote).**
- ✓ **Great for logging, notifications, or running scripts.**
- ✓ **Can be triggered on create, update, or destroy.**
- ✓ **Use carefully—local system dependencies might affect reproducibility.**

Would you like an example tailored to your use case? 🚀