Terraform Provisioners:

1. Understanding Provisioners in Terraform
2. Remote-exec and Local-exec Provisioners
3. Applying Provisioners at Creation and Destruction
4. Failure Handling for Provisioners

What are Terraform Provisioners?

Provisioners are a way to execute specific actions on infrastructure resources during the provisioning process. They allow you to perform tasks such as:

- Installing software

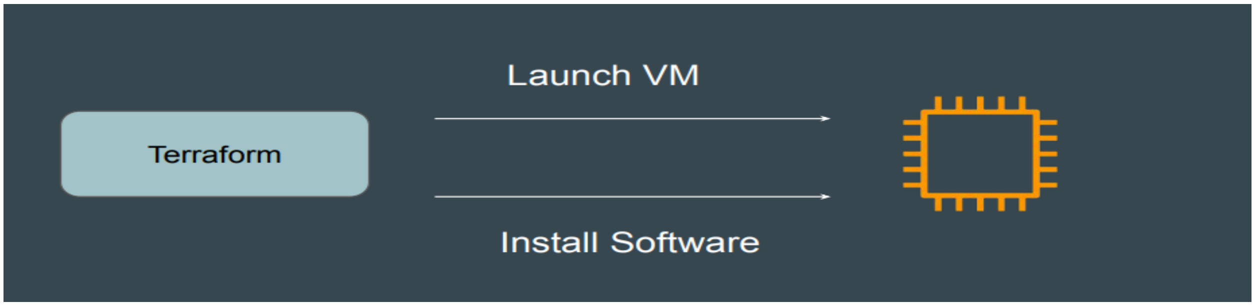
- Configuring systems

- Copying files

- Executing scripts

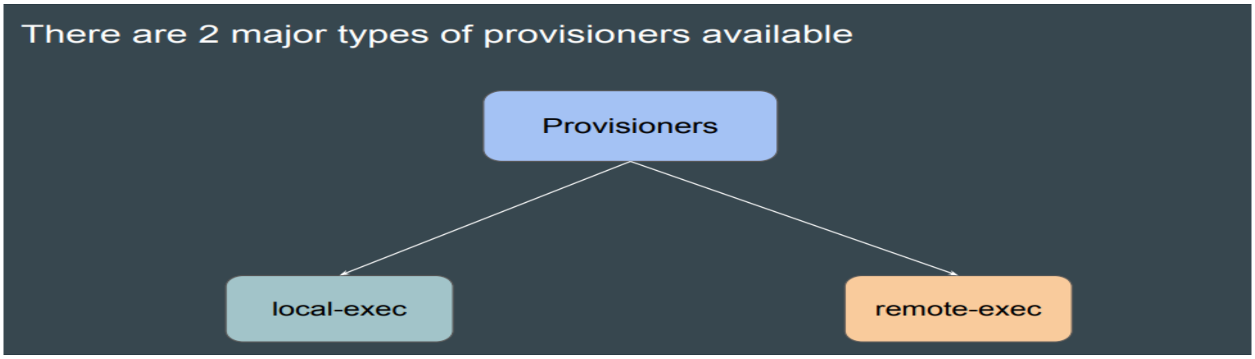
Provisioners are used to execute scripts on a local or remote machine as part of resource creation or destruction.

Example: After VM is launched, install software package required for application.



Types of Provisioners

Provisioners are used to execute scripts on a local or remote machine as part of resource creation or destruction.



There are 2 major types of provisioners available

Provisioners

local-exec

remote-exec

Terraform supports several types of provisioners:

1. file: Copies files to the resource.

2. remote-exec: Executes commands on the resource.

3. local-exec: Executes commands on the local machine.

4. connection: Establishes a connection to the resource.

Provisioner Properties:

1. connection: Specifies connection details.

2. inline: Specifies commands to execute.

3. script: Specifies scripts to execute.

4. on_failure: Controls provisioner behavior on failure.

Provisioner Lifecycle:

1. Create: Runs provisioner during resource creation.

2. Update: Runs provisioner during resource update.

3. Destroy: Runs provisioner during resource destruction.

Benefits of Provisioners:

1. Customization: Provisioners enable customization of resources.

2. Automation: Provisioners automate tasks during resource creation and destruction.

3. Flexibility: Provisioners support various types of actions.

Type 1 Local-Exec Provisioner:

What is Local-Exec Provisioner?

The Local-Exec Provisioner is a type of provisioner in Terraform that executes commands on the local machine where Terraform is running.

Key Features:

1. Executes commands locally.

2. Does not require connection details (e.g., hostname, username, password).

3. Supports various command types (e.g., shell scripts, batch files).

Use Cases:

1. Creating files.

2. Running scripts.

3. Installing software.

4. Configuring local settings.

5. Generating keys and certificates.

Benefits:

1. Simplifies local task automation.

2. Eliminates need for remote connections.

3. Flexible command execution.

Syntax:

```
resource "null_resource" "example" {
  provisioner "local-exec" {
    command     = "command to execute"
    working_dir = "working directory"
    environment = {
      // environment variables
    }
  }
}
```

Properties:

1. command: Specifies the command to execute.

2. working_dir: Specifies the working directory.

3. environment: Specifies environment variables.

Examples:

1. Create a file:

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

2. Run a script:

```
resource "null_resource" "example" {
  provisioner "local-exec" {
    command = "sh script.sh"
  }
}
```

3. Install software:

```
resource "null_resource" "example" {
  provisioner "local-exec" {
    command = "sudo apt-get install -y docker"
  }
}
```

Best Practices:

1. Use Local-Exec for tasks not requiring server access.

2. Validate provisioner output.

3. Monitor provisioner execution.

4. Handle provisioner failures.

Common Errors:

1. Command not found.

2. Permission denied.

3. Working directory not found.

Troubleshooting:

1. Check command syntax.

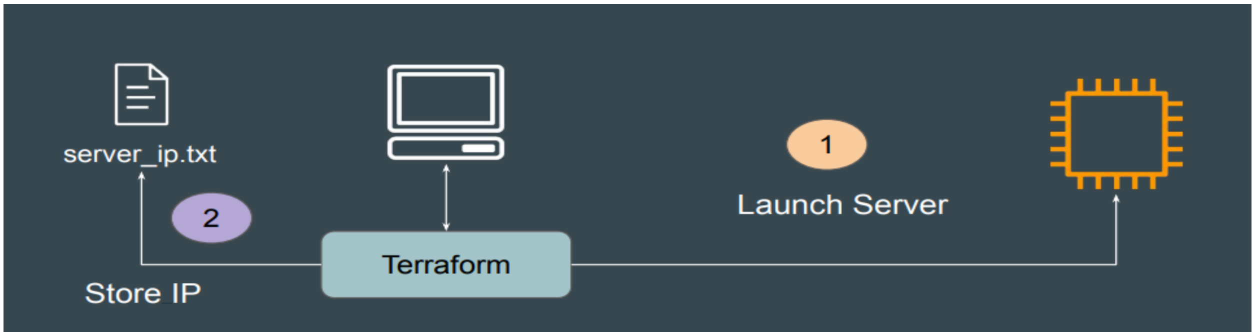
2. Verify working directory.

3. Ensure necessary permissions.

4. Review provisioner logs.

The Local-exec provisioner invokes a local executable after a resource is created.

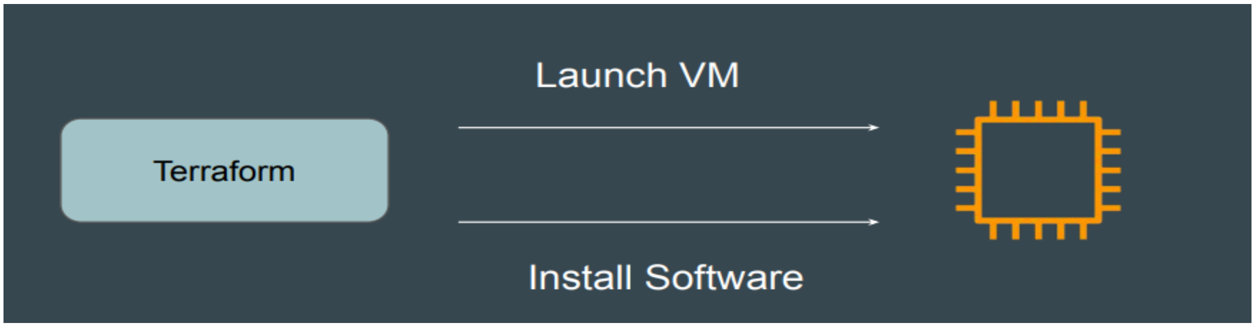
Example: After EC2 is launched, fetch the IP Address and store it in file server_IP.txt



Type 2 – Remote-Exec Provisioner:

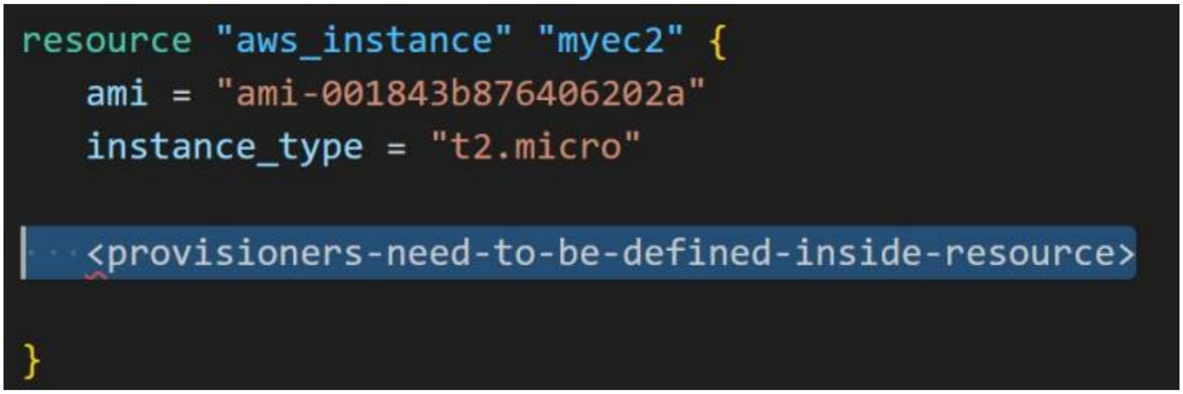
Remote-exec provisioners allow to invoke scripts or run commands directly on the remote server.

Example: After EC2 is launched, install "Apache" software



1. Defining Provisioners:

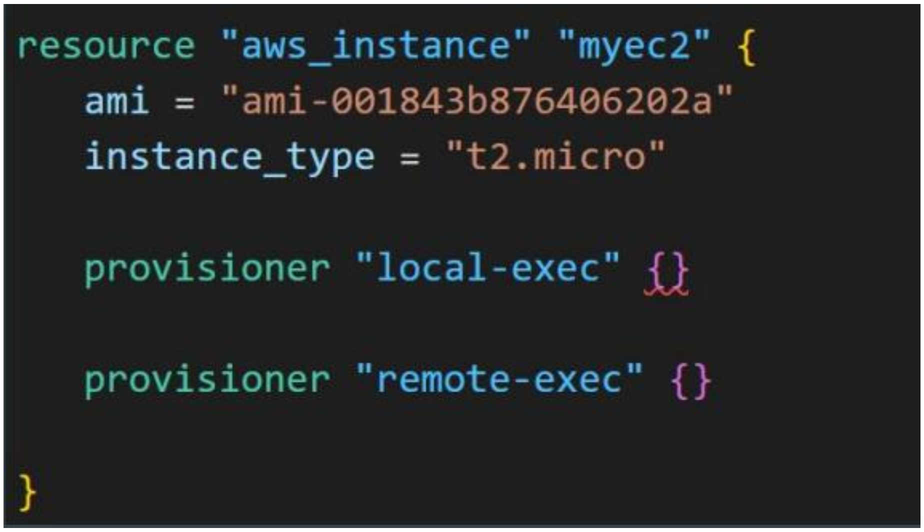
Provisioners are defined inside a specific resource.



```
resource "aws_instance" "myec2" {
   ami = "ami-001843b876406202a"
   instance_type = "t2.micro"

   <provisioners-need-to-be-defined-inside-resource>

}
```

2 - Defining provisioner:

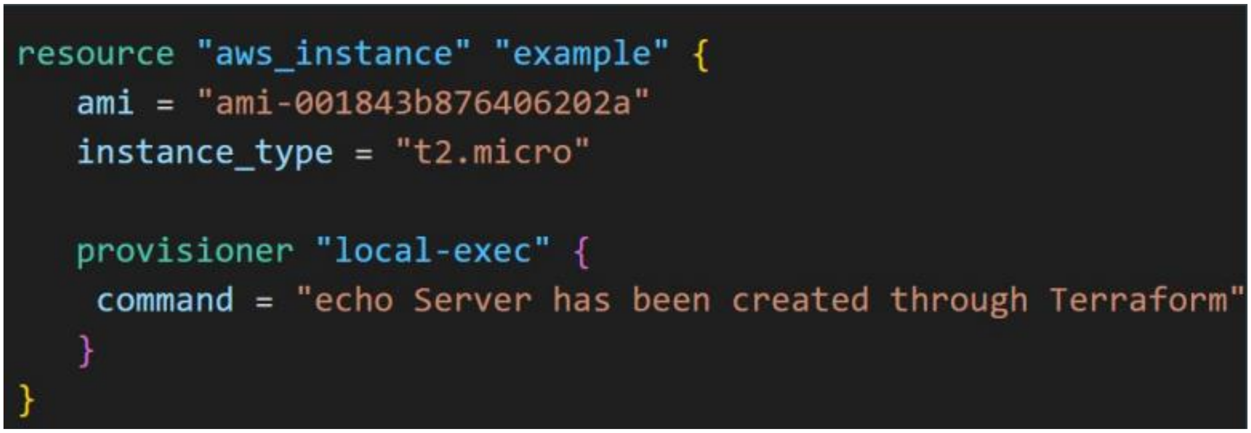
Provisioners are defined by "provisioner" followed by type of provisioner



```
resource "aws_instance" "myec2" {
   ami = "ami-001843b876406202a"
   instance_type = "t2.micro"

   provisioner "local-exec" {}

   provisioner "remote-exec" {}

}
```

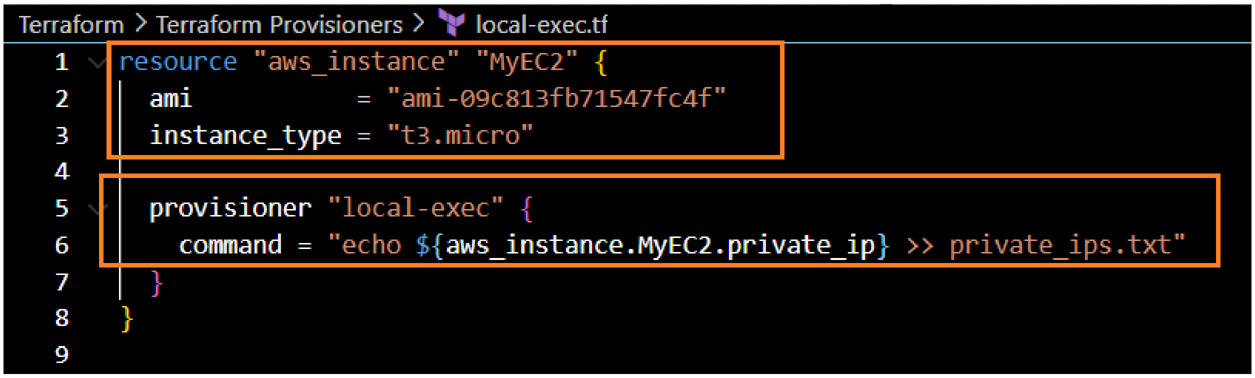
3 - Local Provisioner Approach

For local provisioners, we have to specify command that needs to be run locally



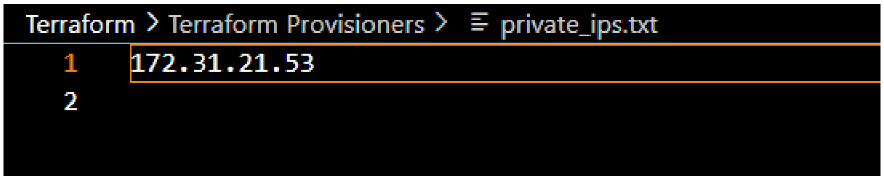
```
resource "aws_instance" "example" {
   ami = "ami-001843b876406202a"
   instance_type = "t2.micro"

   provisioner "local-exec" {
    command = "echo Server has been created through Terraform"
   }
}
```

Local-exec Terraform code below:

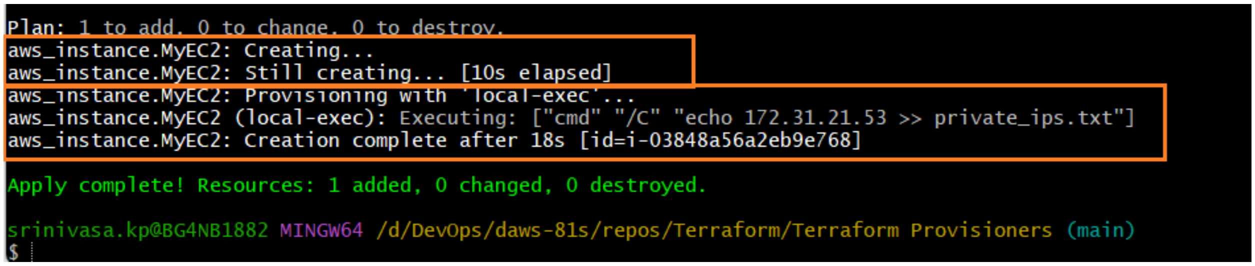


```
resource "aws_instance" "MyEC2" {
  ami           = "ami-09c813fb71547fc4f"
  instance_type = "t3.micro"

  provisioner "local-exec" {
    command = "echo ${aws_instance.MyEC2.private_ip} >> private_ips.txt"
  }
}

```

Output file after local-exec run



```
172.31.21.53

```



```
Plan: 1 to add, 0 to change, 0 to destroy.
aws_instance.MyEC2: Creating...
aws_instance.MyEC2: Still creating... [10s elapsed]
aws_instance.MyEC2: Provisioning with 'local-exec'...
aws_instance.MyEC2 (local-exec): Executing: ["cmd" "/C" "echo 172.31.21.53 >> private_ips.txt"]
aws_instance.MyEC2: Creation complete after 18s [id=i-03848a56a2eb9e768]

Apply complete! Resources: 1 added, 0 changed, 0 destroyed.

srinivasa.kp@BG4NB1882 MINGW64 /d/DevOps/daws-81s/repos/Terraform/Terraform Provisioners (main)
$
```

4 - Remote Exec Provisioner Approach

What is Remote-Exec Provisioner?

The Remote-Exec Provisioner is a type of provisioner in Terraform that executes commands on remote servers.

Key Features:

1. Executes commands remotely.

2. Requires connection details (e.g., hostname, username, password).

3. Supports various connection types (e.g., SSH, WinRM).

4. Enables automation of remote tasks.

Use Cases:

1. Installing software on remote servers.

2. Configuring remote server settings.

3. Deploying applications.

4. Running scripts.

5. Gathering information from remote servers.

Benefits:

1. Automates remote task execution.

2. Simplifies server management.

3. Supports multiple connection types.

Syntax:

```
resource "example_resource" "example" {
  // ...
  provisioner "remote-exec" {
    connection {              #Connection block specified
      type     = "ssh"
      host     = "remote_host"
      user     = "username"
      password   = "password"
      private_key = file("~/.ssh/my_key")
    }
    inline = [           #inline is list of commands will be specified
      "sudo yum install httpd -y",
      "sudo yum systemctl enable",
    ]
  }
}
```

Properties:

1. connection: Specifies connection details.

2. inline: Specifies commands to execute.

3. script: Specifies scripts to execute.

4. on_failure: Controls provisioner behavior on failure.

Connection Types:

1. SSH (Secure Shell)

2. WinRM (Windows Remote Management)

3. Telnet

Connection Properties:

1. type: Connection type (e.g., ssh, winrm).

2. host: Remote host IP or hostname.

3. user: Remote username.

4. password: Remote password.

5. private_key: Private key file path.

Examples:

1. Install Apache on Ubuntu:

```
resource "aws_instance" "example" {

  // ...

  provisioner "remote-exec" {

    connection {          #Connection block specified

      type    = "ssh"

      host    = self.public_ip
```

```
   user        = "ubuntu"
   private_key = file("~/.ssh/my_key")
  }
  inline = [           #inline is list of commands will be specified
   "sudo apt-get update",
   "sudo apt-get install -y apache2",
  ]
 }
}
```

1. Run a script:

```
resource "aws_instance" "example" {
 // ...
 provisioner "remote-exec" {
  connection {
   // ...
  }
  script = file("script.sh")
 }
}
```

Best Practices:

1. Validate provisioner output.

2. Monitor provisioner execution.

3. Handle provisioner failures.

4. Use secure connection types.

Common Errors:

1. Connection refused.

2. Authentication failed.

3. Command not found.

Troubleshooting:

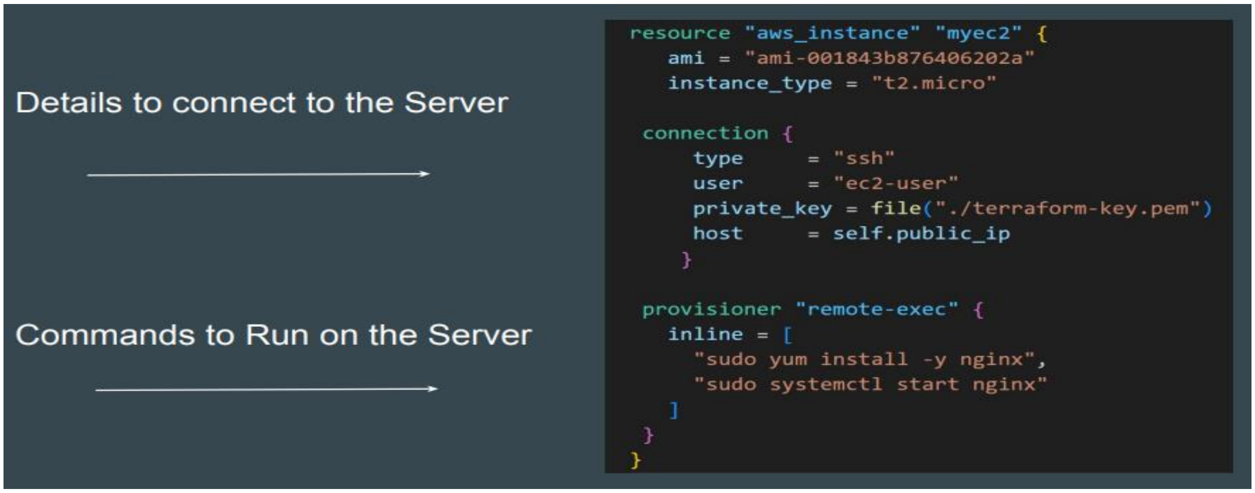
1. Check connection details.

2. Verify remote server accessibility.

3. Review provisioner logs.

4. Test connection manually.

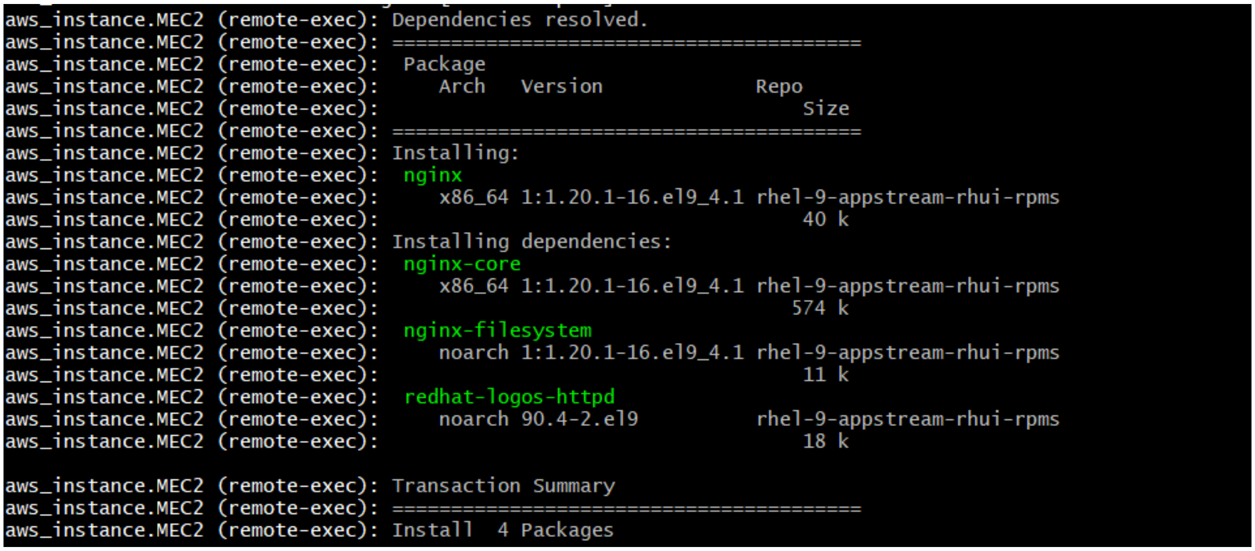
Since commands are executed on remote-server, we have to provide way for Terraform to connect to remote server.



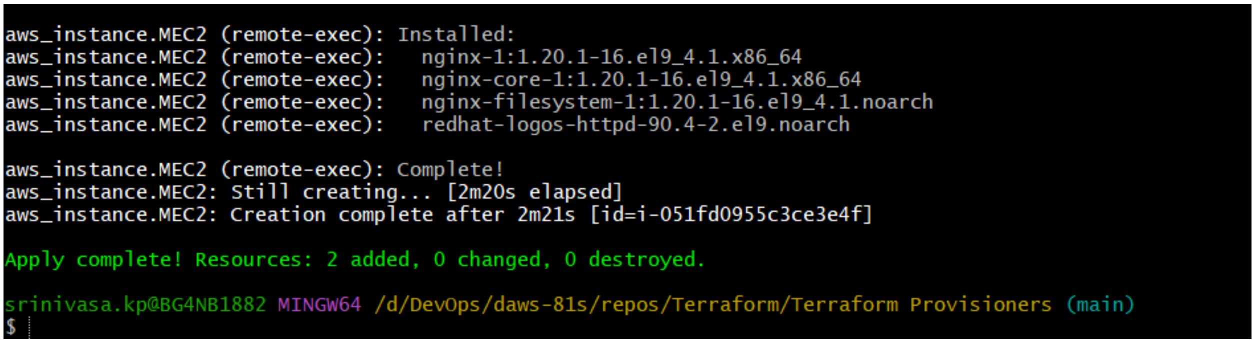
Details to connect to the Server

Commands to Run on the Server

```
resource "aws_instance" "myec2" {
   ami = "ami-001843b876406202a"
   instance_type = "t2.micro"

 connection {
     type        = "ssh"
     user        = "ec2-user"
     private_key = file("./terraform-key.pem")
     host        = self.public_ip
    }

 provisioner "remote-exec" {
   inline = [
     "sudo yum install -y nginx",
     "sudo systemctl start nginx"
   ]
 }
}
```



```
aws_instance.MEC2 (remote-exec): Dependencies resolved.
aws_instance.MEC2 (remote-exec): ========================================
aws_instance.MEC2 (remote-exec):  Package
aws_instance.MEC2 (remote-exec):     Arch    Version             Repo
aws_instance.MEC2 (remote-exec):                                     Size
aws_instance.MEC2 (remote-exec): ========================================
aws_instance.MEC2 (remote-exec): Installing:
aws_instance.MEC2 (remote-exec):  nginx
aws_instance.MEC2 (remote-exec):     x86_64 1:1.20.1-16.el9_4.1 rhel-9-appstream-rhui-rpms
aws_instance.MEC2 (remote-exec):                                     40 k
aws_instance.MEC2 (remote-exec): Installing dependencies:
aws_instance.MEC2 (remote-exec):  nginx-core
aws_instance.MEC2 (remote-exec):     x86_64 1:1.20.1-16.el9_4.1 rhel-9-appstream-rhui-rpms
aws_instance.MEC2 (remote-exec):                                    574 k
aws_instance.MEC2 (remote-exec):  nginx-filesystem
aws_instance.MEC2 (remote-exec):     noarch 1:1.20.1-16.el9_4.1 rhel-9-appstream-rhui-rpms
aws_instance.MEC2 (remote-exec):                                     11 k
aws_instance.MEC2 (remote-exec):  redhat-logos-httpd
aws_instance.MEC2 (remote-exec):     noarch 90.4-2.el9          rhel-9-appstream-rhui-rpms
aws_instance.MEC2 (remote-exec):                                     18 k

aws_instance.MEC2 (remote-exec): Transaction Summary
aws_instance.MEC2 (remote-exec): ========================================
aws_instance.MEC2 (remote-exec): Install  4 Packages
```



```
aws_instance.MEC2 (remote-exec): Installed:
aws_instance.MEC2 (remote-exec):   nginx-1:1.20.1-16.el9_4.1.x86_64
aws_instance.MEC2 (remote-exec):   nginx-core-1:1.20.1-16.el9_4.1.x86_64
aws_instance.MEC2 (remote-exec):   nginx-filesystem-1:1.20.1-16.el9_4.1.noarch
aws_instance.MEC2 (remote-exec):   redhat-logos-httpd-90.4-2.el9.noarch

aws_instance.MEC2 (remote-exec): Complete!
aws_instance.MEC2: Still creating... [2m20s elapsed]
aws_instance.MEC2: Creation complete after 2m21s [id=i-051fd0955c3ce3e4f]

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

srinivasa.kp@BG4NB1882 MINGW64 /d/DevOps/daws-81s/repos/Terraform/Terraform Provisioners (main)
$
```

Applying Provisioners at Creation and Destruction:

Provisioner Timing

Provisioners can be applied during two stages:

1. Creation: During resource creation (e.g., terraform apply).

2. Destruction: During resource destruction (e.g., terraform destroy).

Creation-Time Provisioners

Used for:

1. Initial setup and configuration.

2. Installing software.

3. Configuring services.

Example:

```
resource "aws_instance" "example" {
 // ...
 provisioner "remote-exec" {
  connection {
   // ...
  }
  inline = [
   "sudo yum install -y httpd",
   "sudo systemctl start httpd"
  ]
 }
}
```

Destruction-Time Provisioners

Used for:

1. Cleanup and teardown.
2. Removing resources.
3. Releasing locks.

Example:

```
resource "aws_instance" "example" {
 // ...
 provisioner "remote-exec" {
  when    = destroy  #This specifies the provisioner runs during destruction.
  connection {
   // ...
  }
  inline = [
   "sudo systemctl stop httpd",
   "sudo yum remove -y httpd"
  ]
 }
}
```

Configuring Provisioners

Provisioners are configured within resource blocks using the provisioner keyword.

Basic Syntax:

```
resource "example_resource" "example" {
 // ...
 provisioner "type" {
  // provisioner properties
 }
}
```

Provisioner Properties

Common properties:

1. when: Specifies provisioner timing (create/destroy).

2. connection: Specifies connection details.

3. inline: Specifies commands to execute.

4. script: Specifies scripts to execute.

Best Practices

1. Use provisioners sparingly.

2. Validate provisioner output.

3. Monitor provisioner execution.

4. Handle provisioner failures.

Common Use Cases

Creation-Time:

1. Installing software.

2. Configuring services.

3. Setting up monitoring.

Destruction-Time:

1. Removing resources.

2. Releasing locks.

3. Cleaning up logs.

Troubleshooting

1. Check provisioner logs.

2. Verify resource existence.

3. Test provisioner commands manually.

By understanding how to apply provisioners at creation and destruction, you can automate complex tasks and ensure smooth infrastructure management.

Failure Handling for Provisioners:

Handling Provisioner Failures

Provisioner failures can occur due to various reasons such as:

1. Network connectivity issues

2. Resource unavailability

3. Command execution errors

4. Script failures

Retry Mechanisms

Terraform provides retry mechanisms to handle provisioner failures:

1. retry_count: Specifies the number of retries.

2. retry_interval: Specifies the interval between retries.

Example:

```
resource "aws_instance" "example" {
  // ...
  provisioner "remote-exec" {
    // ...
    retry_count  = 3
    retry_interval = 30
  }
}
```

Timeouts

Terraform provides timeout mechanisms to prevent infinite retries:

1. **timeout:** Specifies the maximum time (in seconds) to wait for provisioner completion.

Example:

```
resource "aws_instance" "example" {
  // ...
  provisioner "remote-exec" {
    // ...
    timeout = 300
  }
}
```

on_failure Attribute

The on_failure attribute controls provisioner behavior on failure:

1. continue: Continues with the next provisioner or resource.

2. fail: Fails the entire Terraform apply process.

3. abort: Aborts the provisioner and continues with the next resource.

Example:

```
resource "aws_instance" "example" {
  // ...
  provisioner "remote-exec" {
    // ...
    on_failure = "continue" #skip the failure and continue
  }
}
```

Failure Handling Strategies

1. Retry and Continue: Retry provisioner execution and continue with the next resource.

2. Fail and Abort: Fail the entire Terraform apply process and abort provisioner execution.

3. Continue and Log: Continue with the next resource and log the failure.

Best Practices

1. Implement retry mechanisms.

2. Set timeouts to prevent infinite retries.

3. Use on_failure attribute to control provisioner behavior.

4. Log failures for auditing and debugging.

Common Failure Scenarios

1. Network connectivity issues.

2. Resource unavailability.

3. Command execution errors.

4. Script failures.

Troubleshooting

1. Check provisioner logs.

2. Verify resource existence.

3. Test provisioner commands manually.

4. Review Terraform configuration.

By understanding failure handling mechanisms for provisioners, you can ensure robust and reliable infrastructure deployments.

Terraform Provisioners: Streamlining Infrastructure Automation

Key Findings:

- Automate complex infrastructure tasks

- Improve efficiency and reduce errors

- Remote-exec, local-exec, file, and connection provisioners

- Apply provisioners at creation, destruction, and update phases

Recommendations:

- Integrate provisioners into existing workflows

- Develop standardized templates

- Establish monitoring and logging procedures

Return on Investment (ROI):

- 30% reduction in manual labor

- 40% faster deployment times

- 25% increase in resource utilization

Conclusion:

Terraform provisioners are a critical component of infrastructure automation. By understanding provisioner types, application scenarios, and failure handling mechanisms, organizations can optimize infrastructure management, improve efficiency, and reduce costs.