TASKS ON TERRAFORM

- ❖ **NAME: S.MUZZAMMIL HUSSAIN**
- ❖ **DATE: 23/07/2025**
- ❖ **BATCH: 11**
- ❖ **NO.OF TASKS: 1**

Task 1. Explore the null provider

The **null provider** in Terraform is a special utility provider created by HashiCorp that offers constructs which **intentionally do nothing** on their own. Despite this, it serves as a useful tool to orchestrate complex behaviours or work around certain limitations in Terraform configurations.

What is the null provider and null_resource?

- The **null provider** itself doesn't manage any real infrastructure. Instead, it exposes the null_resource type.
- A **null_resource** is a resource that creates no actual infrastructure but can be used to **run provisioners, local commands, or trigger actions** based on changes in inputs or dependencies.
- It's typically used to orchestrate workflows that don't fit neatly into other resource types or to run arbitrary scripts before or after other resources get applied.

Common Use Cases for the null provider/null_resource

- **Run local or remote commands** during Terraform apply using provisioners like local-exec or remote-exec.
- Create **dependency relationships** where some code must run after or before a particular resource without creating actual infrastructure.
- Workarounds where Terraform lacks native features to perform procedural or imperative operations.
- **Trigger re-execution** of commands based on input changes via the triggers argument.

Why use the null provider?

- It helps **orchestrate tricky behaviors** that can't be expressed declaratively through existing resources.
- Provides hooks to **run custom scripts or commands** fully integrated with Terraform's dependency graph.
- Can be used to **chain multiple Terraform modules or configurations**, e.g., by using a null_resource to run terraform apply on another folder before provisioning dependent resources15.

Caveats and Considerations

- Usage of null provider can **make configuration harder to understand** and maintain, so it should be used sparingly and only when necessary
- Alternatives like the new terraform_data resource (introduced in Terraform 1.4) may replace some use cases of null_resource by being built-in and easier to reason about.
- Null provider and null_resource require the null provider plugin to be present, unlike many other Terraform features which are built-in.

Summary

Aspect	Description
Provider Name	null (HashiCorp null provider)
Function	Provides null_resource that intentionally does nothing but can run provisioners or trigger actions based on input changes.
Use Cases	Run local/remote commands, orchestrate resource dependencies, workaround Terraform limitations, chain configurations.
Trigger Mechanism	triggers argument to re-run on changes.
Alternatives	terraform_data (Terraform 1.4+), built-in data resource with similar capabilities.
Caveats	Increases complexity; use only when necessary.

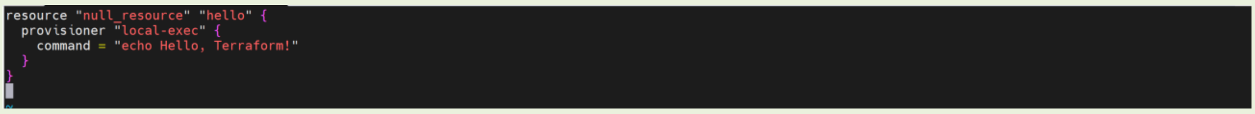
In essence, the null provider is a Terraform utility to support **imperative actions and workflow orchestration without managing real infrastructure**, enabling more flexible Terraform workflows.

Simple Example of a null_resource in Terraform

Here's a minimal example of how to use **null_resource** with the null provider in Terraform:

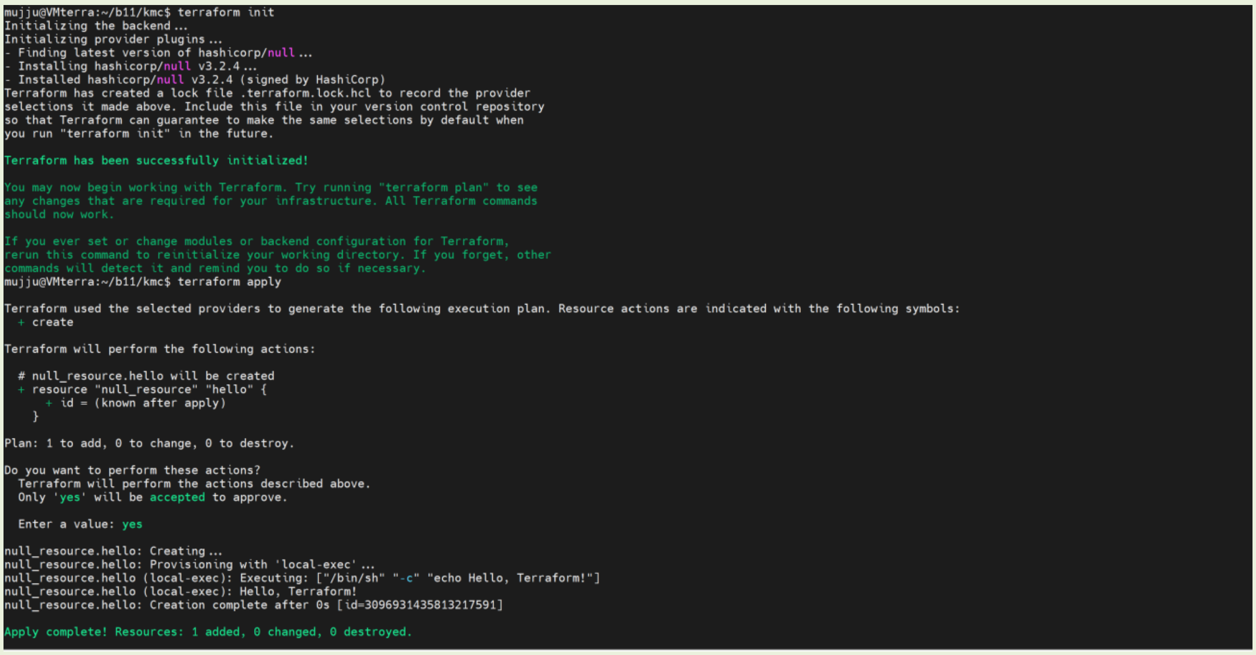
```
resource "null_resource" "hello" {
  provisioner "local-exec" {
    command = "echo Hello, Terraform!"
  }
}
```

Create a file named **main.tf** using **vi main.tf** and enter the following and save using esc:wq



```
resource "null_resource" "hello" {
  provisioner "local-exec" {
    command = "echo Hello, Terraform!"
  }
}
```

Run the commands **terraform init** and **terraform apply**



```
mujju@VMterra:~/b11/kmc$ terraform init
Initializing the backend...
Initializing provider plugins...
- Finding latest version of hashicorp/null...
- Installing hashicorp/null v3.2.4...
- Installed hashicorp/null v3.2.4 (signed by HashiCorp)
Terraform has created a lock file .terraform.lock.hcl to record the provider
selections it made above. Include this file in your version control repository
so that Terraform can guarantee to make the same selections by default when
you run "terraform init" in the future.

Terraform has been successfully initialized!

You may now begin working with Terraform. Try running "terraform plan" to see
any changes that are required for your infrastructure. All Terraform commands
should now work.

If you ever set or change modules or backend configuration for Terraform,
rerun this command to reinitialize your working directory. If you forget, other
commands will detect it and remind you to do so if necessary.
mujju@VMterra:~/b11/kmc$ terraform apply

Terraform used the selected providers to generate the following execution plan. Resource actions are indicated with the following symbols:
  + create

Terraform will perform the following actions:

  # null_resource.hello will be created
  + resource "null_resource" "hello" {
      + id = (known after apply)
    }

Plan: 1 to add, 0 to change, 0 to destroy.

Do you want to perform these actions?
  Terraform will perform the actions described above.
  Only 'yes' will be accepted to approve.

  Enter a value: yes

null_resource.hello: Creating...
null_resource.hello: Provisioning with 'local-exec'...
null_resource.hello (local-exec): Executing: ["/bin/sh" "-c" "echo Hello, Terraform!"]
null_resource.hello (local-exec): Hello, Terraform!
null_resource.hello: Creation complete after 0s [id=3096931435813217591]

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

Explanation:

- This configuration defines a resource of type null_resource and names it hello.
- Inside, it uses a local-exec provisioner to run a command on your local machine whenever the resource is created or modified.
- The command echo Hello, Terraform! will display the message in your console during terraform apply.

Use Case:
This is useful for running one-off scripts, tests, or arbitrary commands that should be integrated into your Terraform workflow without creating any real infrastructure resources.

You can expand this by using the triggers argument to make the resource rerun whenever specific input changes, but the core idea is as shown above.

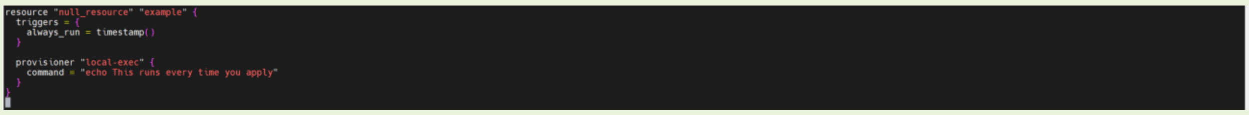
Making it re-run on changes

If you want the null_resource to execute every time or when something specific changes, you add the triggers argument, for example:

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

  provisioner "local-exec" {
    command = "echo This runs every time you apply"
  }
}
```

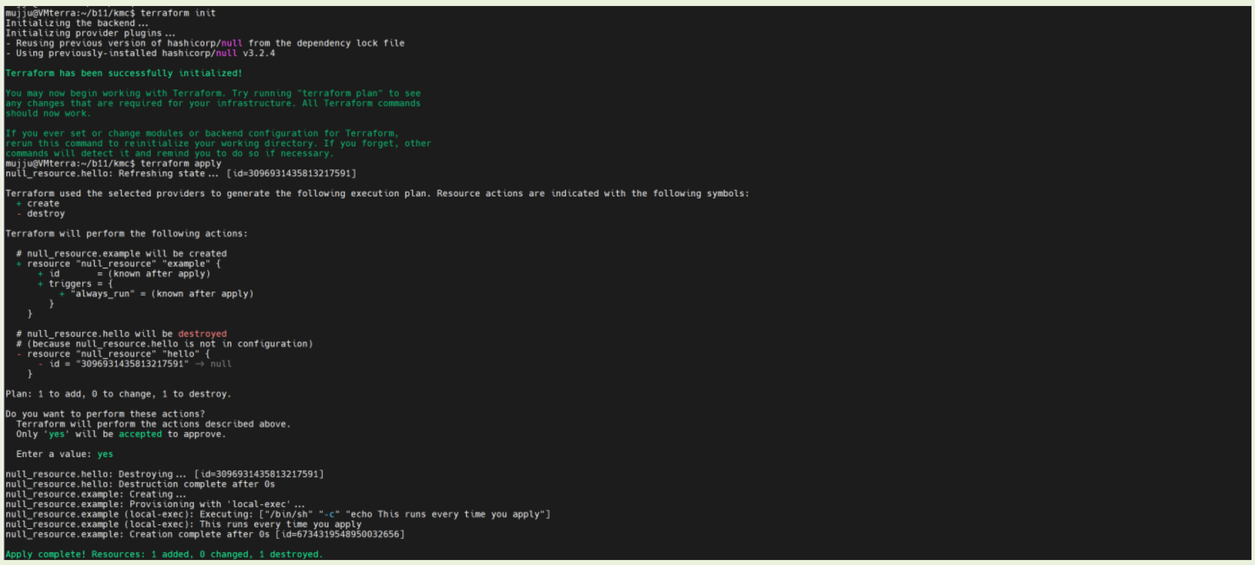
Create a file named **main.tf** using **vi main.tf** and enter the following and save using esc:wq



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

  provisioner "local-exec" {
    command = "echo This runs every time you apply"
  }
}
```

Run the commands **terraform init** and **terraform apply**



```
mujju@VMterra:~/b11/kmc$ terraform init
Initializing the backend...
Initializing provider plugins...
- Reusing previous version of hashicorp/null from the dependency lock file
- Using previously-installed hashicorp/null v3.2.4

Terraform has been successfully initialized!

You may now begin working with Terraform. Try running "terraform plan" to see
any changes that are required for your infrastructure. All Terraform commands
should now work.

If you ever set or change modules or backend configuration for Terraform,
rerun this command to reinitialize your working directory. If you forget, other
commands will detect it and remind you to do so if necessary.
mujju@VMterra:~/b11/kmc$ terraform apply
null_resource.hello: Refreshing state... [id=3096931435813217591]

Terraform used the selected providers to generate the following execution plan. Resource actions are indicated with the following symbols:
  + create
  - destroy

Terraform will perform the following actions:

  # null_resource.example will be created
  + resource "null_resource" "example" {
      + id       = (known after apply)
      + triggers = {
          + "always_run" = (known after apply)
        }
    }

  # null_resource.hello will be destroyed
  # (because null_resource.hello is not in configuration)
  - resource "null_resource" "hello" {
      - id = "3096931435813217591" -> null
    }

Plan: 1 to add, 0 to change, 1 to destroy.

Do you want to perform these actions?
  Terraform will perform the actions described above.
  Only 'yes' will be accepted to approve.

  Enter a value: yes

null_resource.hello: Destroying... [id=3096931435813217591]
null_resource.hello: Destruction complete after 0s
null_resource.example: Creating...
null_resource.example: Provisioning with 'local-exec'...
null_resource.example (local-exec): Executing: ["/bin/sh" "-c" "echo This runs every time you apply"]
null_resource.example (local-exec): This runs every time you apply
null_resource.example: Creation complete after 0s [id=6734319548950032656]

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

- Here, the null_resource reruns on every terraform apply because the timestamp always changes.
- Alternatively, you can use triggers based on file hashes or other variables to rerun only when inputs change.

Summary

Feature	Description
null_resource	Creates no real infrastructure
Purpose	Run local or remote commands during apply
Default behavior	Executes command once on first apply
triggers	Optional map to force re-creation and re-run
Common provisioners	local-exec, remote-exec

This simple pattern is widely used in Terraform workflows to perform imperative tasks integrated into the deployment lifecycle without managing real infrastructure resources.

Simple Example: Terraform Null Data Source

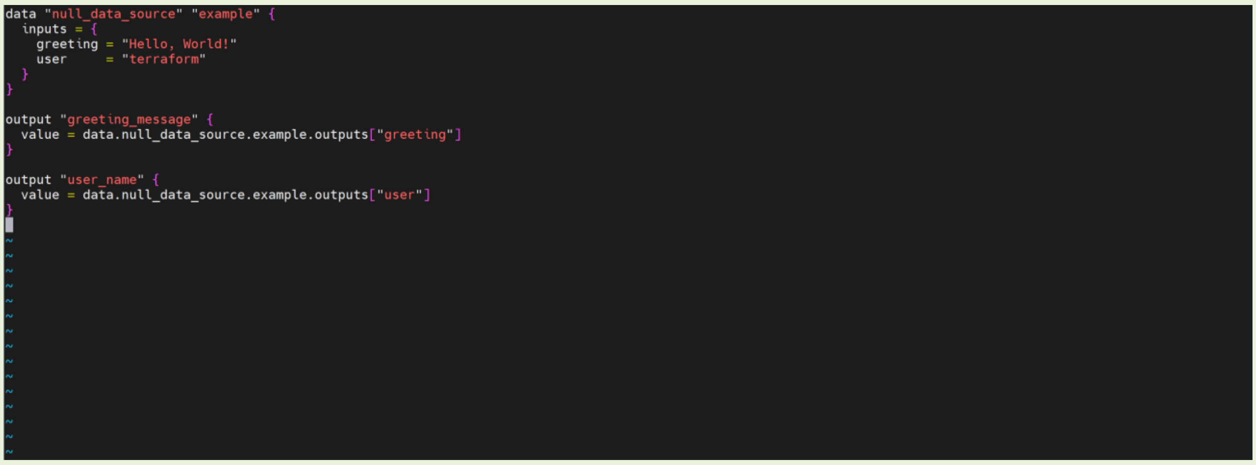
The null_data_source was historically used in Terraform to pass intermediate values or make simple data available for reference elsewhere in your configuration. It's now considered legacy (locals and the terraform_data resource are usually preferred), but here's how you use it:

```
data "null_data_source" "example" {
  inputs = {
    greeting = "Hello, World!"
    user     = "terraform"
  }
}

output "greeting_message" {
  value = data.null_data_source.example.outputs["greeting"]
}

output "user_name" {
  value = data.null_data_source.example.outputs["user"]
}
```

Create a file named **main.tf** using **vi main.tf** and enter the following and save using esc:wq

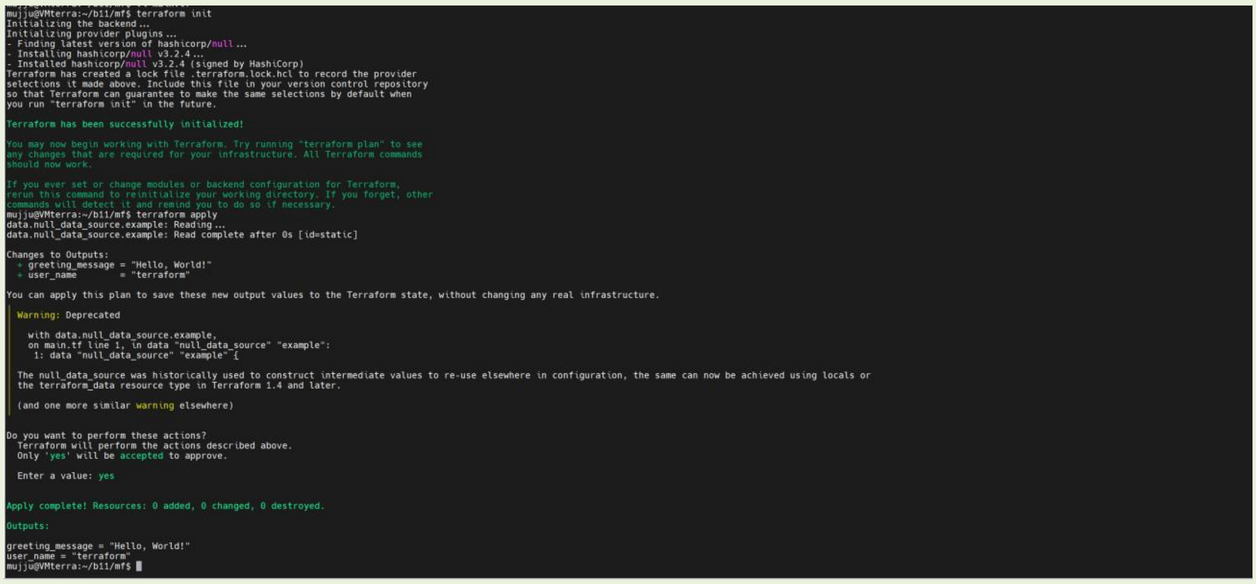


```
data "null_data_source" "example" {
  inputs = {
    greeting = "Hello, World!"
    user     = "terraform"
  }
}

output "greeting_message" {
  value = data.null_data_source.example.outputs["greeting"]
}

output "user_name" {
  value = data.null_data_source.example.outputs["user"]
}
```

Run the commands **terraform init** and **terraform apply**



```
mujju@VMterra:~/b11/mf$ terraform init
Initializing the backend...
Initializing provider plugins...
- Finding latest version of hashicorp/null...
- Installing hashicorp/null v3.2.4...
- Installed hashicorp/null v3.2.4 (signed by HashiCorp)
Terraform has created a lock file .terraform.lock.hcl to record the provider
selections it made above. Include this file in your version control repository
so that Terraform can guarantee to make the same selections by default when
you run "terraform init" in the future.

Terraform has been successfully initialized!

You may now begin working with Terraform. Try running "terraform plan" to see
any changes that are required for your infrastructure. All Terraform commands
should now work.

If you ever set or change modules or backend configuration for Terraform,
rerun this command to reinitialize your working directory. If you forget, other
commands will detect it and remind you to do so if necessary.
mujju@VMterra:~/b11/mf$ terraform apply
data.null_data_source.example: Reading...
data.null_data_source.example: Read complete after 0s [id=static]

Changes to Outputs:
  + greeting_message = "Hello, World!"
  + user_name        = "terraform"

You can apply this plan to save these new output values to the Terraform state, without changing any real infrastructure.

│ Warning: Deprecated
│
│   with data.null_data_source.example,
│   on main.tf line 1, in data "null_data_source" "example":
│    1: data "null_data_source" "example" {
│
│ The null_data_source was historically used to construct intermediate values to re-use elsewhere in configuration, the same can now be achieved using locals or
│ the terraform_data resource type in Terraform 1.4 and later.
│
│ (and one more similar warning elsewhere)

Do you want to perform these actions?
  Terraform will perform the actions described above.
  Only 'yes' will be accepted to approve.

  Enter a value: yes


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

Outputs:

greeting_message = "Hello, World!"
user_name = "terraform"
mujju@VMterra:~/b11/mf$ █
```

How This Works

- The null_data_source named example accepts a map of input strings.
- The outputs attribute makes those input values available elsewhere in your configuration.
- The output blocks export these values so you see them after running terraform apply.

What to Expect

- When you run terraform apply, you'll see both "Hello, World!" and "terraform" as outputs.
- Nothing is created or changed in any cloud provider—this is purely data-handling.

Output Example

Output Name	Output Value
greeting_message	Hello, World!
user_name	terraform

When Should You Use This?

- For new code, **prefer local values** (locals { ... }) or the terraform_data resource.
- Use null_data_source only for compatibility with older modules or legacy codebases[1].

Quick Comparison Table

Feature	Null Data Source	Local Values
Typing	Strings only	Any type
Syntax	Verbose	Concise
Preferred for new code?	No	Yes

This example demonstrates a straightforward way to store and access data using a null data source in Terraform