TASKS ON TERRAFORM

- **NAME: S.MUZZAMMIL HUSSAIN**
- **DATE: 19/07/2025**
- **BATCH: 11**
- **NO.OF TASKS: 1**

Task 1. Terraform Subsequent apply, destroy

Create a file using **vi file.tf** and enter the contents as follows

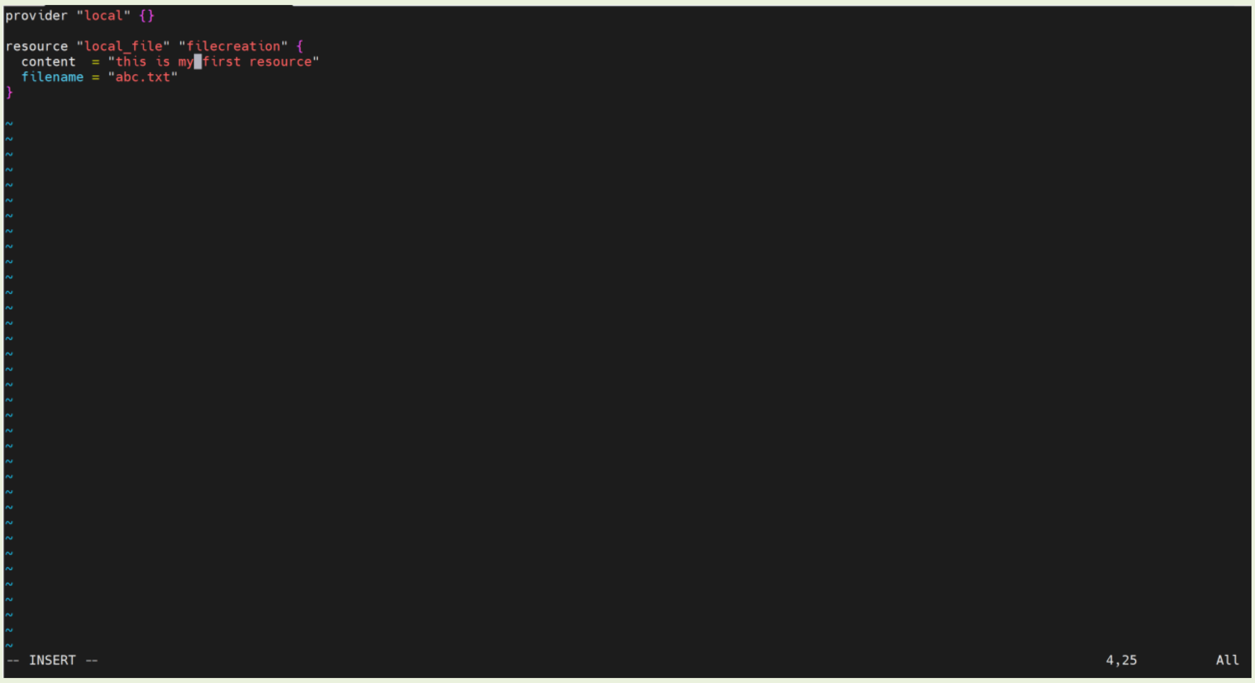
Provider “local” {}

resource "local_file" "filecreation" {

content = "this is my first resource"

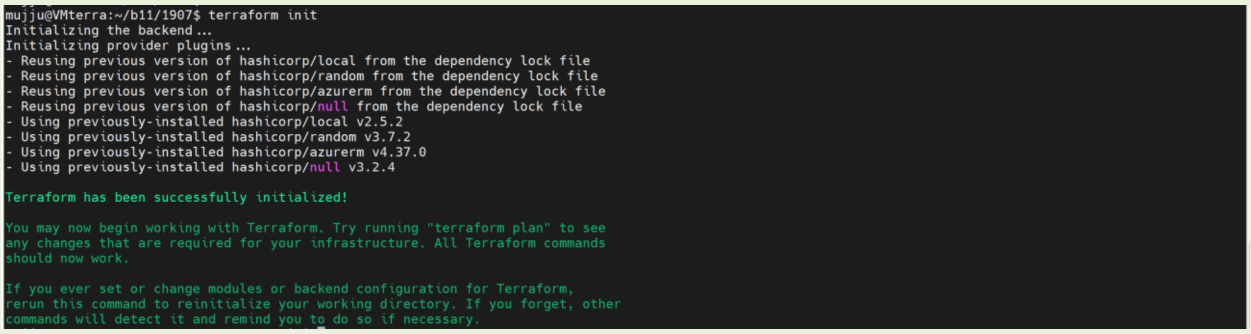
filename = "abc.txt"

}



```
provider "local" {}

resource "local_file" "filecreation" {
  content  = "this is my first resource"
  filename = "abc.txt"
}
~
-- INSERT --                                                    4,25          All
```

Run the command **terraform init**



```
mujju@VMterra:~/b11/1907$ terraform init
Initializing the backend...
Initializing provider plugins...
- Reusing previous version of hashicorp/local from the dependency lock file
- Reusing previous version of hashicorp/random from the dependency lock file
- Reusing previous version of hashicorp/azurerm from the dependency lock file
- Reusing previous version of hashicorp/null from the dependency lock file
- Using previously-installed hashicorp/local v2.5.2
- Using previously-installed hashicorp/random v3.7.2
- Using previously-installed hashicorp/azurerm v4.37.0
- 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.
```

Terraform "init" Flows Explained

The image outlines the internal workflow when running the terraform init command in Terraform. Here's how these steps apply within Terraform:

1. CLI to Core to Instance Site

- **CLI (Command Line Interface)**: The user runs terraform init from the terminal.
- **Core**: The command is received by Terraform's core engine.
- **Instansite (likely "instantiate")**: Terraform prepares to set up the core components of the working directory.

2. Core Creates Workspace

- Terraform's core engine creates or prepares the workspace directory where all state and configuration files will be stored.
- If workspaces are used, it ensures the correct workspace environment is ready.

3. Core Reads Configuration and Lists Providers/Modules

- Terraform core scans all .tf files in the current directory.
- It parses the configuration to determine:
  - Which **providers** (like AWS, Azure, Google Cloud) are needed.
  - Which **modules** (pre-packaged configurations) are required.

4. Core Connects to Registry and Downloads Providers

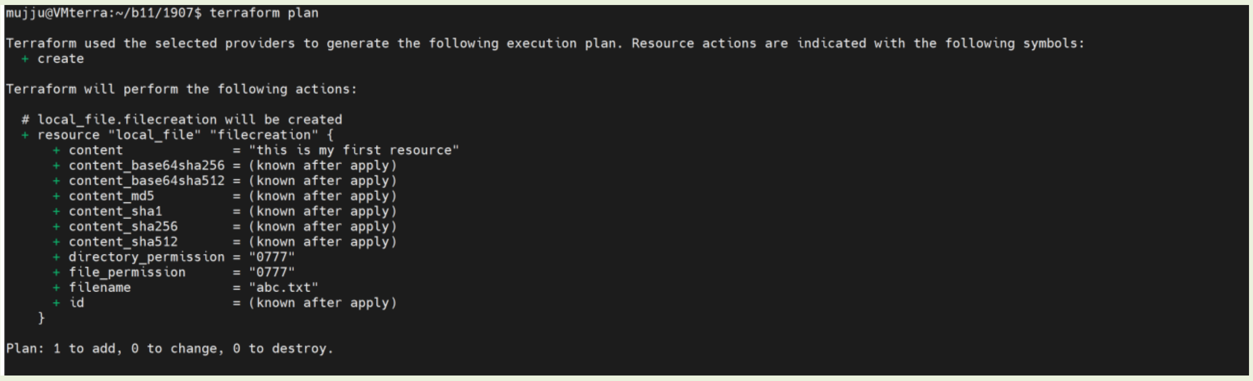
- After identifying the required providers, Terraform connects to the provider **registry** (like the public Terraform Registry).
- It **downloads** the necessary provider binaries to the local .terraform directory.

Summary Table

Step	Description
CLI Init	User runs terraform init in the terminal.
Core Setup	Initializes the workspace (and creates directory).
Read Configs	Parses .tf files to list providers and modules.
Download	Downloads providers from registry if not already present locally.

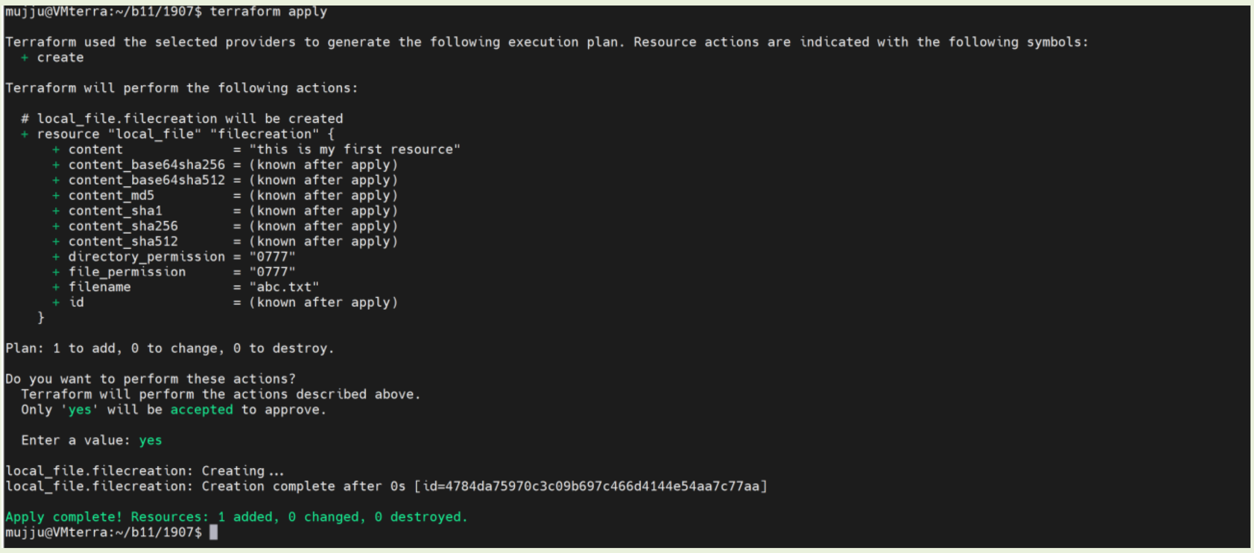
In essence, **terraform init** prepares our local environment by reading configurations, setting up the workspace, and ensuring all required providers and modules are ready for use.

Run the command **terraform plan**



```
mujju@VMterra:~/b11/1907$ terraform plan

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:

  # local_file.filecreation will be created
  + resource "local_file" "filecreation" {
      + content              = "this is my first resource"
      + content_base64sha256 = (known after apply)
      + content_base64sha512 = (known after apply)
      + content_md5          = (known after apply)
      + content_sha1         = (known after apply)
      + content_sha256       = (known after apply)
      + content_sha512       = (known after apply)
      + directory_permission = "0777"
      + file_permission      = "0777"
      + filename             = "abc.txt"
      + id                   = (known after apply)
    }

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

Run the command **terraform apply**



```
mujju@VMterra:~/b11/1907$ 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:

  # local_file.filecreation will be created
  + resource "local_file" "filecreation" {
      + content              = "this is my first resource"
      + content_base64sha256 = (known after apply)
      + content_base64sha512 = (known after apply)
      + content_md5          = (known after apply)
      + content_sha1         = (known after apply)
      + content_sha256       = (known after apply)
      + content_sha512       = (known after apply)
      + directory_permission = "0777"
      + file_permission      = "0777"
      + filename             = "abc.txt"
      + 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

local_file.filecreation: Creating...
local_file.filecreation: Creation complete after 0s [id=4784da75970c3c09b697c466d4144e54aa7c77aa]

Apply complete! Resources: 1 added, 0 changed, 0 destroyed.
mujju@VMterra:~/b11/1907$ 
```

Terraform "apply" Flows Explained

Here's an **in-depth, step-by-step breakdown of the internal execution flow of terraform apply** when using the **local provider to create a local file**, as you described (e.g., creating abc.txt using a file.tf configuration):

1. CLI ➔ Core: Run terraform apply

- Execute terraform apply from our terminal.
- The **Terraform core** starts the orchestration process. It is responsible for reading your configs and directing providers to take action.

2. Core : Reads the Configuration

- The core loads our file.tf (and any other .tf files in the directory).
- It parses out the resources (like local_file), providers, variables, and outputs.
- The configuration is validated for syntax and logical correctness.

3. Core ➔ Local Provider: Resource Creation Request

- The core recognizes a resource block such as:

  ```
  resource "local_file" "filecreation" {
    content  = "this is my first resource"
    filename = "abc.txt"
  }
  ```

- For each resource, core tells the **local provider** to create or update it.

4. Local Provider ➔ Resource: Creates the File

- The **local provider plugin** takes the request and acts on the system: here, it creates a new file named abc.txt with the specified content on your local filesystem.
- If the file already exists and the content is different, the provider will update it; if nothing has changed, it will leave it as is.

5. Provider ➔ Core: Returns Resource Status

- The local provider sends status and details back to the core:
    - Was the file created or updated successfully?
    - What is the path, checksum, etc.?

6. Core ➔ State File: Updates State

- The Terraform core updates the terraform.tfstate file.
- It records that local_file.example now exists, with metadata needed to track and manage this file in future operations.
- The state file is crucial—it enables Terraform to know what it manages, and makes subsequent actions (change, destroy, recreate) possible.

7. CLI Output: User Feedback

- Terraform displays a summary:

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

- If there are outputs, they're shown here as well.

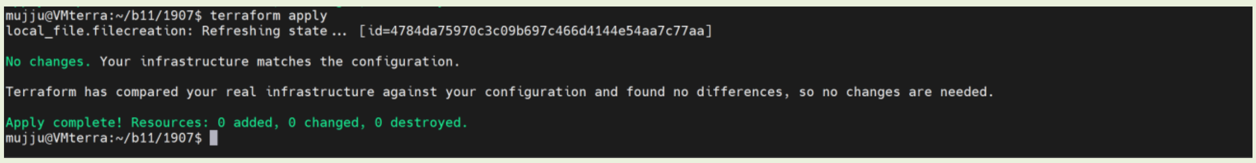
Step by step explanation:

- We run terraform apply.
- Terraform parses the config, requests the local provider to create abc.txt.
- The provider creates abc.txt with your content.
- Status is reported to the core, which records this in state.
- You see the message confirming the resource was created.

Key Takeaways

- **Core** manages the overall orchestration and state tracking.
- **Providers** (here, local) execute the low-level operations, such as file creation.
- **State file** (terraform.tfstate) is essential for tracking what exists, preventing accidental duplication or loss of resources.
- **Workflow repeats for each resource**—the same flow would trigger for each additional file or change you declare in your configuration.

Run the command **terraform apply** again



```
mujju@VMterra:~/b11/1907$ terraform apply
local_file.filecreation: Refreshing state... [id=4784da75970c3c09b697c466d4144e54aa7c77aa]

No changes. Your infrastructure matches the configuration.

Terraform has compared your real infrastructure against your configuration and found no differences, so no changes are needed.

Apply complete! Resources: 0 added, 0 changed, 0 destroyed.
mujju@VMterra:~/b11/1907$ 
```

When we perform a **subsequent terraform apply** meaning we run the command again without changing our configuration or the underlying target (local, cloud, etc.) the internal workflow operates as follows:

Step-by-Step Process for a Subsequent Apply

1. **Terraform Reads State and Config**
   - Terraform loads our existing terraform.tfstate file, which reflects the last known state of all resources (e.g., our local file abc.txt if already created).
   - It parses our current .tf files (such as file.tf describing the local_file resource) to determine the desired state.
2. **Terraform Plans Changes**
   - By default, a plan step compares:
     - The desired state (from our configuration)
     - The current state (from the state file, optionally refreshed from the real infrastructure in our case, the local file system)
   - If nothing has changed in our configuration and our managed resources have not been altered outside of Terraform, the plan detects no actions required.
3. **User Review (Interactive Mode)**
   - Terraform produces an execution plan, which will say:

     No changes. Your infrastructure matches the configuration.

   - The prompt for confirmation appears, but since there are no proposed changes, this is informational.
4. **Apply Step**
   - Since there are no differences:
     - **No new resources are created**.
     - **No existing resources are changed or destroyed**.
     - **No provider API calls or file operations are made**.
   - The state file is left unchanged

Example for the Local Provider

Assume your file.tf contains:

```
resource "local_file" "filecreation" {
  content  = "this is my first resource"
  filename = "abc.txt"
}
```

- **First apply:**
  - Creates abc.txt with the given content and records it in state.
- **Subsequent apply:**
  - Terraform checks that abc.txt is present and unchanged.
  - Output will be:

    No changes. Your infrastructure matches the configuration.

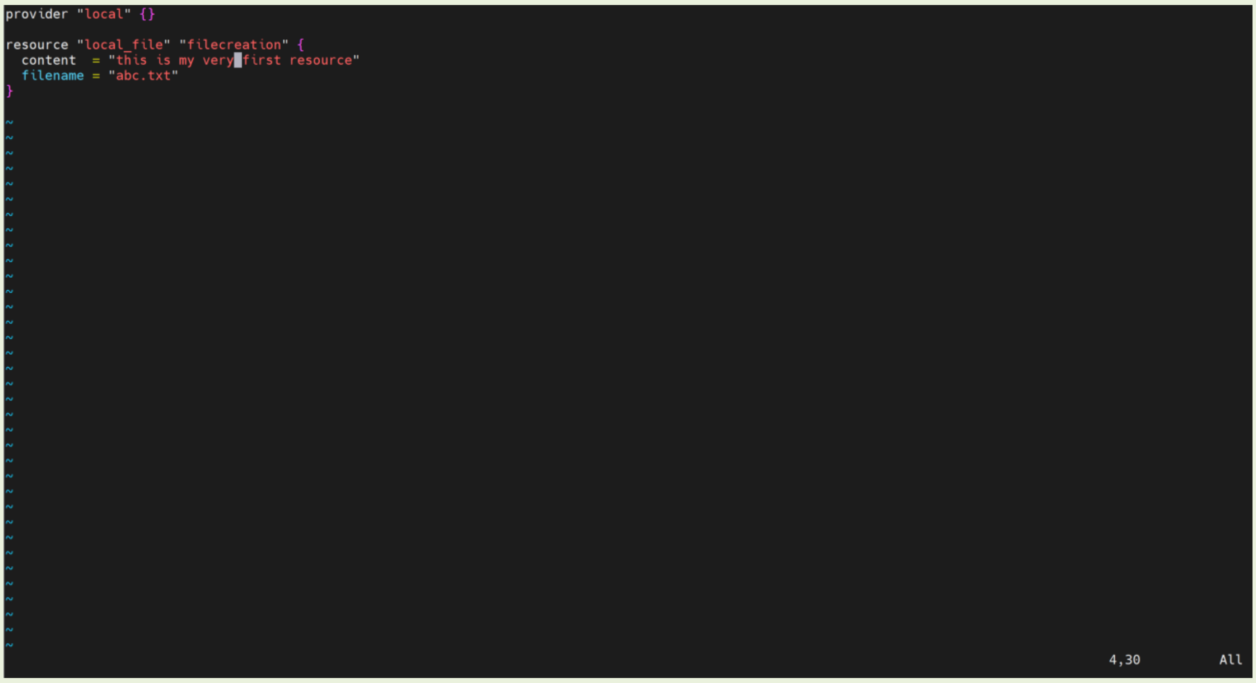
- Nothing is written, deleted, or modified.

Why This Matters

- **Idempotency:**
  Running terraform apply repeatedly with unmodified configuration is *safe*; Terraform will not perform duplicate actions.
- **Drift Detection:**
  If the file was changed or deleted outside Terraform, or if you edit the resource content in your .tf, then terraform apply will detect a difference and plan an update or recreation to restore the desired state.
- **Core Workflow:**
  This applies to any provider and resource type, including local files, AWS EC2 instances, etc

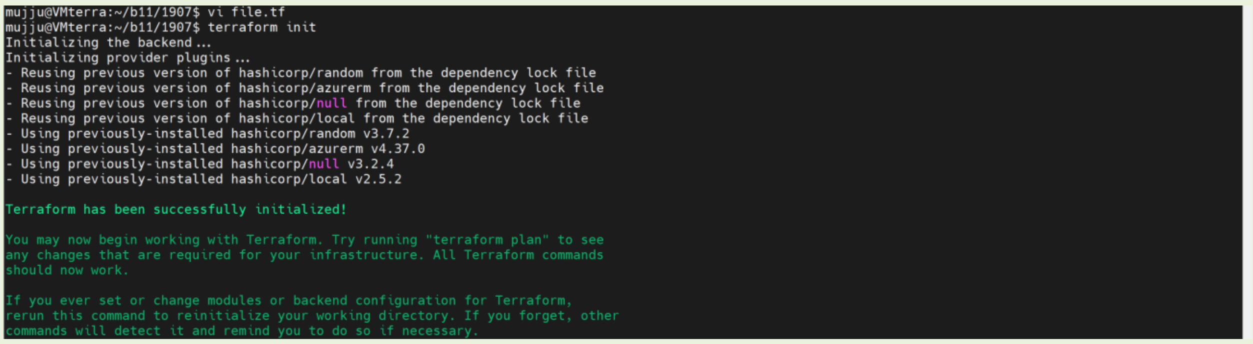
Summary:
On a subsequent terraform apply with no changes, Terraform simply checks for differences, finds none, and makes no modifications—your local file (or any managed resource) remains as-is, and you get a confirmation message indicating the infrastructure matches your configuration.

We make some changes in the content of file.tf as follows



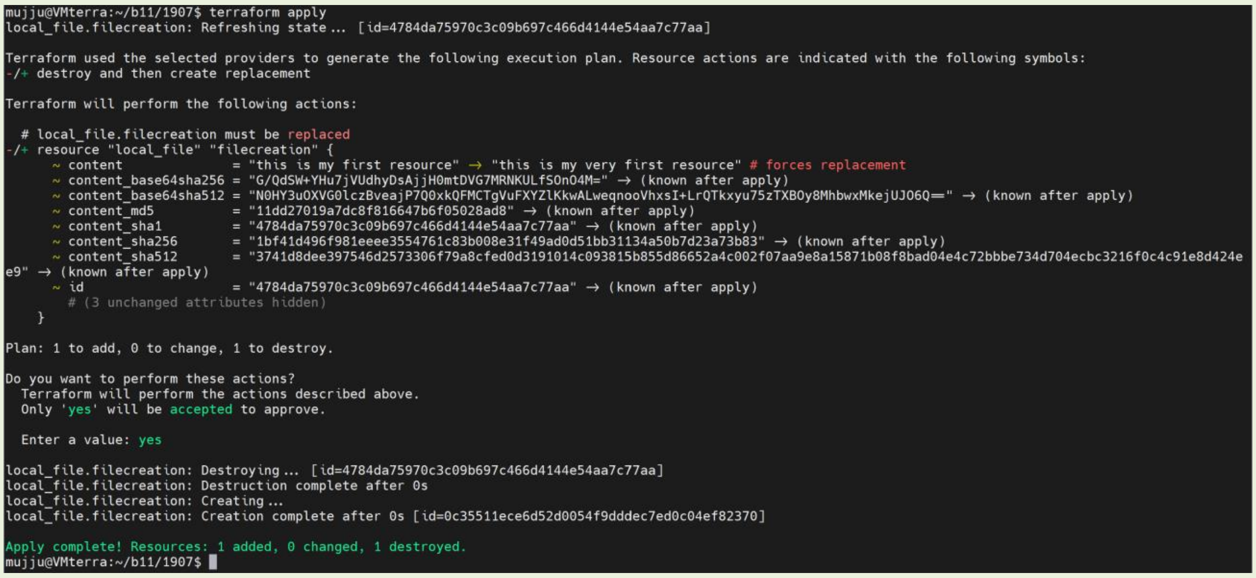
```
provider "local" {}

resource "local_file" "filecreation" {
  content  = "this is my very first resource"
  filename = "abc.txt"
}

~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
                                                                              4,30          All
```

Run the command **terraform init**



```
mujju@VMterra:~/b11/1907$ vi file.tf
mujju@VMterra:~/b11/1907$ terraform init
Initializing the backend...
Initializing provider plugins...
- Reusing previous version of hashicorp/random from the dependency lock file
- Reusing previous version of hashicorp/azurerm from the dependency lock file
- Reusing previous version of hashicorp/null from the dependency lock file
- Reusing previous version of hashicorp/local from the dependency lock file
- Using previously-installed hashicorp/random v3.7.2
- Using previously-installed hashicorp/azurerm v4.37.0
- Using previously-installed hashicorp/null v3.2.4
- Using previously-installed hashicorp/local v2.5.2

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.
```

Run the command **terraform apply**



```
mujju@VMterra:~/b11/1907$ terraform apply
local_file.filecreation: Refreshing state... [id=4784da75970c3c09b697c466d4144e54aa7c77aa]

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

Terraform will perform the following actions:

  # local_file.filecreation must be replaced
-/+ resource "local_file" "filecreation" {
      ~ content              = "this is my first resource" → "this is my very first resource" # forces replacement
      ~ content_base64sha256 = "G/QdSW+YHu7jVUdhyDsAjjH0mtDVG7MRNKULfSOnO4M=" → (known after apply)
      ~ content_base64sha512 = "N0HY3uOXVG0lczBveajP7Q0xkQFMCTgVuFXYZlKkwALweqnooVhxsI+LrQTkxyu75zTXBOy8MhbwxMkejUJ06Q==" → (known after apply)
      ~ content_md5          = "11dd27019a7dc8f816647b6f05028ad8" → (known after apply)
      ~ content_sha1         = "4784da75970c3c09b697c466d4144e54aa7c77aa" → (known after apply)
      ~ content_sha256       = "1bf41d496f981eeee3554761c83b008e31f49ad0d51bb31134a50b7d23a73b83" → (known after apply)
      ~ content_sha512       = "3741d8dee397546d2573306f79a8cfed0d3191014c093815b855d86652a4c002f07aa9e8a15871b08f8bad04e4c72bbbe734d704ecbc3216f0c4c91e8d424e
e9" → (known after apply)
      ~ id                   = "4784da75970c3c09b697c466d4144e54aa7c77aa" → (known after apply)
        # (3 unchanged attributes hidden)
    }

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

local_file.filecreation: Destroying... [id=4784da75970c3c09b697c466d4144e54aa7c77aa]
local_file.filecreation: Destruction complete after 0s
local_file.filecreation: Creating...
local_file.filecreation: Creation complete after 0s [id=0c35511ece6d52d0054f9dddec7ed0c04ef82370]

Apply complete! Resources: 1 added, 0 changed, 1 destroyed.
mujju@VMterra:~/b11/1907$
```

We have **edited the content** for our local_file resource in our Terraform configuration (for example, updating the content field in our .tf file), the workflow for terraform apply changes as follows:

1. **Terraform Reads State and Configuration**
    - Terraform loads the existing terraform.tfstate, which contains the previous content of your local file (e.g., abc.txt).
    - It reads your updated .tf file, noting the *new* content you want for abc.txt.
2. **Terraform Plans a Change**
    - It compares the state (old content) with your configuration (new content).
    - Since the content has changed, Terraform generates a plan to update the local_file resource.
    - The plan preview will show:

      ```
      ~ resource "local_file" "abc" {
          ~ content = "old content" -> "new content"
        }
      ```

      The tilde (~) indicates an update—in this case, the file content will change.

3. **User Confirmation**
    - Terraform prompts you to approve the change. You can review exactly what will be modified.
4. **Apply Phase**
    - Once approved, Terraform instructs the local provider to update the file on disk.
    - The file abc.txt is **overwritten with the new content** specified in your configuration.
5. **State Update**
    - The new content (and its checksum) is recorded in the state file.
    - Subsequent applies will use this updated state for further comparisons.

File is immediately updated on disk to match what’s in our configuration, and Terraform’s state remains accurate for future runs.

Run terraform destroy

Running **terraform destroy** initiates the destruction workflow in Terraform for all resources defined in your current configuration and tracked in the state file.

Step-by-Step Execution

1. **Pre-Check and Planning**
    - Terraform loads your .tf files (such as file.tf) and reads the current terraform.tfstate file.
    - It compares the resources tracked in state (for example, your local file abc.txt) with your configuration.
    - A destruction plan is created, showing all resources scheduled for removal. You are shown a summary like:

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

2. **User Confirmation**
    - You are prompted to review the plan and confirm destruction (unless running with -auto-approve).
    - The plan details what will be destroyed—here, the resource local_file.abc.
3. **Resource Destruction**
    - After approval, Terraform instructs the local provider to delete the resource. For your case:
        - The file abc.txt is deleted from your local filesystem.
4. **State File Update**
    - Terraform updates the state file, removing all references to the destroyed resource.
    - The state accurately reflects that the resource (the file) no longer exists.
5. **Output Message**
    - Terraform presents a final message:

      Destroy complete! Resources: 1 destroyed.

Illustrative Example

Suppose your configuration includes:

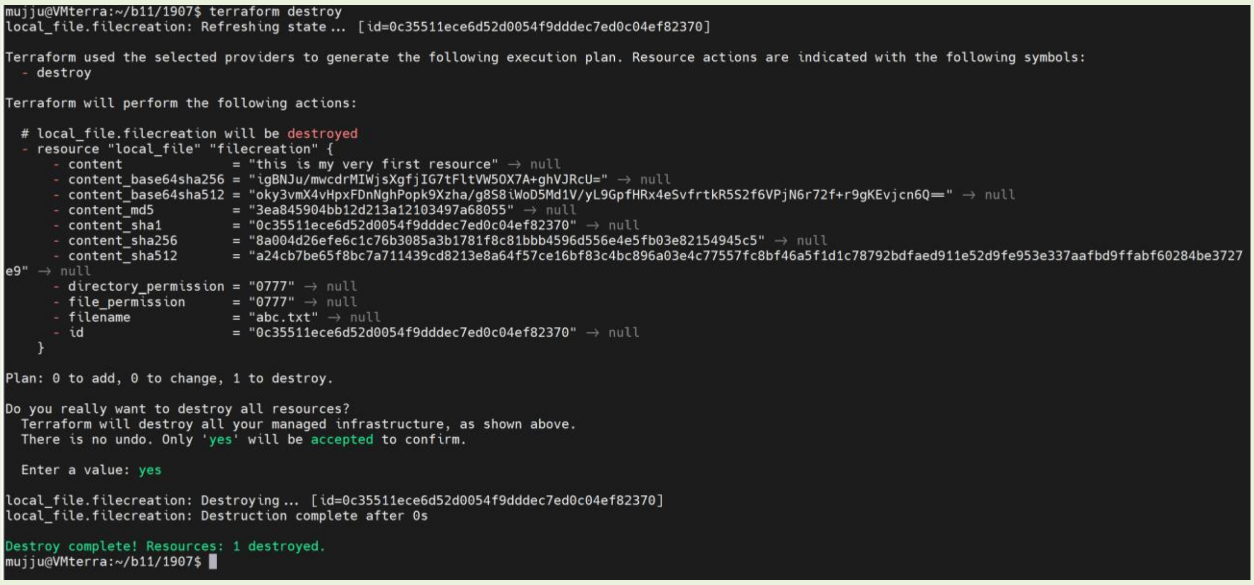
```
resource "local_file" "filecreation" {
  content  = "this is my first resource"
  filename = "abc.txt"
}
```

- After running terraform destroy and confirming, the file abc.txt will be deleted, and Terraform will no longer track or manage this file in the state.

Key Points

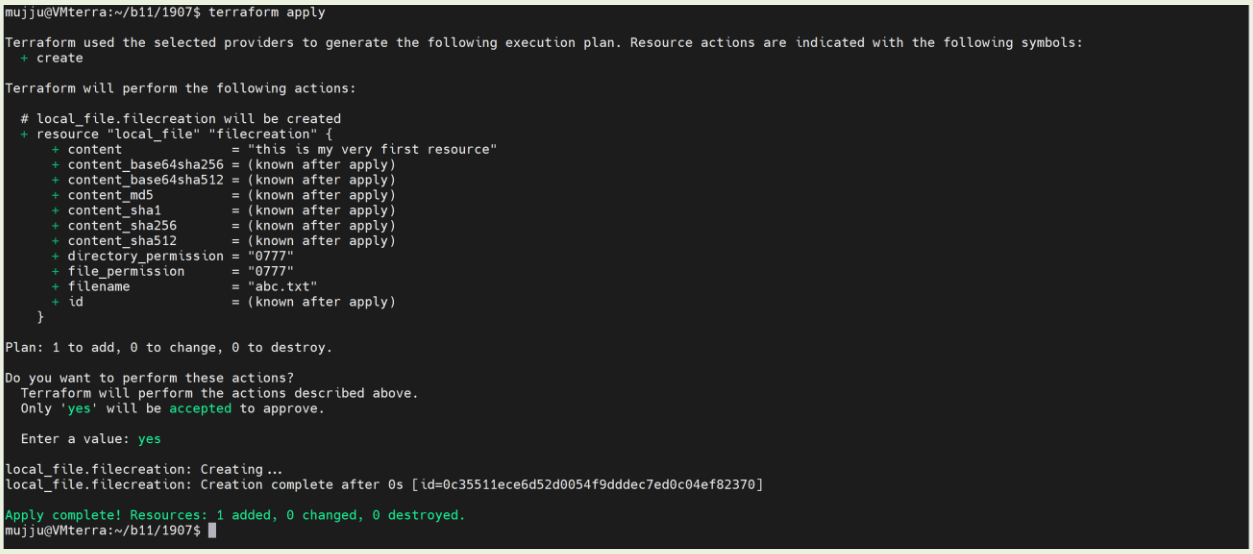
- Destruction is **safe and explicit**: Terraform only destroys resources it tracks in state and that are defined in your configuration.
- **State management**: After destroy, the resource is gone from both your filesystem and from Terraform's perspective.
- If you run terraform apply again with the same configuration, the file will be recreated.

Run the command **terraform destroy**



```
mujju@VMterra:~/b11/1907$ terraform destroy
local_file.filecreation: Refreshing state... [id=0c35511ece6d52d0054f9dddec7ed0c04ef82370]

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

Terraform will perform the following actions:

  # local_file.filecreation will be destroyed
  - resource "local_file" "filecreation" {
      - content              = "this is my very first resource" -> null
      - content_base64sha256 = "igBNJu/mwcdrMIWjsXgfjIG7tFltVW5OX7A+ghVJRcU=" -> null
      - content_base64sha512 = "oky3vmX4vHpxFDnNghPopk9Xzha/g8S8iWoD5Md1V/yL9GpfHRx4eSvfrtkR5S2f6VPjN6r72f+r9gKEvjcn6Q==" -> null
      - content_md5          = "3ea845904bb12d213a12103497a68055" -> null
      - content_sha1         = "0c35511ece6d52d0054f9dddec7ed0c04ef82370" -> null
      - content_sha256       = "8a004d26efe6c1c76b3085a3b1781f8c81bbb4596d556e4e5fb03e82154945c5" -> null
      - content_sha512       = "a24cb7be65f8bc7a711439cd8213e8a64f57ce16bf83c4bc896a03e4c77557fc8bf46a5f1d1c78792bdfaed911e52d9fe953e337aafbd9ffabf60284be3727
e9" -> null
      - directory_permission = "0777" -> null
      - file_permission      = "0777" -> null
      - filename             = "abc.txt" -> null
      - id                   = "0c35511ece6d52d0054f9dddec7ed0c04ef82370" -> null
    }

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

Do you really want to destroy all resources?
  Terraform will destroy all your managed infrastructure, as shown above.
  There is no undo. Only 'yes' will be accepted to confirm.

  Enter a value: yes

local_file.filecreation: Destroying... [id=0c35511ece6d52d0054f9dddec7ed0c04ef82370]
local_file.filecreation: Destruction complete after 0s

Destroy complete! Resources: 1 destroyed.
mujju@VMterra:~/b11/1907$ █
```

If we run **terraform apply** again with the same configuration, the file will be recreated.



```
mujju@VMterra:~/b11/1907$ 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:

  # local_file.filecreation will be created
  + resource "local_file" "filecreation" {
      + content              = "this is my very first resource"
      + content_base64sha256 = (known after apply)
      + content_base64sha512 = (known after apply)
      + content_md5          = (known after apply)
      + content_sha1         = (known after apply)
      + content_sha256       = (known after apply)
      + content_sha512       = (known after apply)
      + directory_permission = "0777"
      + file_permission      = "0777"
      + filename             = "abc.txt"
      + 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

local_file.filecreation: Creating...
local_file.filecreation: Creation complete after 0s [id=0c35511ece6d52d0054f9dddec7ed0c04ef82370]

Apply complete! Resources: 1 added, 0 changed, 0 destroyed.
mujju@VMterra:~/b11/1907$ █
```