TERRAFORM - DETAILED EXPLANATION

WHY TERRAFORM?

HOW TERRAFORM WORKS

As a user: You will configure a Terraform provider and Terraform will talk to the target API.

Process:

Step 1: Configure provider You provide the Terraform provider as AWS

Step 2: Terraform understands Whatever the user has written (configuration files), Terraform will convert this file into API that AWS will understand.

Step 3: Resources created Terraform makes API calls to AWS and creates resources

---

SWITCHING CLOUD PROVIDERS

Tomorrow if you want to change the provider to Azure:

What you do:

1. Modify provider in script file
2. Change resource types
3. Update parameters

What Terraform does: Terraform will handle the rest. It will convert your configuration to Azure API calls.

Key point: If you learn Terraform, once you provide details and provider name, everything will be done by Terraform.

ADVANTAGE 1: MANAGE ANY INFRASTRUCTURE

With Terraform, we can manage any kind of infrastructure.

Supported platforms:

- AWS
- Azure
- Google Cloud
- IBM Cloud
- Oracle Cloud
- VMware
- OpenStack
- Kubernetes
- And 3000+ more providers

Benefits:

- One tool for everything
- Consistent syntax
- Same workflow
- Portable skills

ADVANTAGE 2: INFRASTRUCTURE VISIBILITY

Without Terraform:

- Need to log in to cloud provider console
- Check each service separately
- Difficult to track what exists
- No central view

With Terraform: You need not have to log in to your cloud provider and check what all infrastructure you have created.

How it works:

1. Simply log in to your Terraform machine
2. Look at your state file
3. State file shows everything created

Where is state file stored? State file may be stored in:

- S3 bucket (AWS)
- Azure Storage Container
- Google Cloud Storage
- Any other remote backend

This is called remote backend.

Benefit: You can look at the state file and you will understand what all resources were created. This way you can keep track of your infrastructure.

ADVANTAGE 3: AUTOMATE CHANGES

Manual approach (without Terraform):

1. Log in to cloud console
2. Navigate to resource
3. Click edit
4. Make changes
5. Save
6. Repeat for each resource

Terraform approach: With Terraform, you don't have to manually log in to your infrastructure and make changes.

How it works:

1. Update Terraform file
2. Run terraform apply
3. Changes applied automatically

You can automate changes

ADVANTAGE 4: COLLABORATION

With Terraform, you can collaborate as well.

How collaboration works:

Step 1: Version control Whenever you want to make any change, you can put your Terraform files (except your state file) in a version control system like Git repository.

Step 2: Use version control for collaboration Using this version control system, you can collaborate with your peers.

COLLABORATION EXAMPLE

Scenario: You want to increase the resources of your EC2 instance

Without Terraform:

1. Manually change in AWS console
2. No review process
3. No tracking
4. Risk of errors

With Terraform:

1. Go to Git repository
2. Update your Terraform file
3. Create pull request
4. Ask one of your peers to review it
5. Peer reviews changes
6. Peer approves
7. Merge to main branch
8. Run terraform apply
9. Changes applied

Benefits:

- Code review process
- Track who changed what
- Rollback capability
- Team collaboration
- Audit trail

In this way, you can automate changes with collaboration.

ADVANTAGE 5: STANDARDIZE CONFIGURATIONS

What is standardization? There is a standard that we are maintaining with Terraform files. We are standardizing the way in which we are writing configuration files.

Why important:

- Consistent format
- Team follows same structure
- Easy to understand
- Maintainable
- Reusable

Example standard:

```
project/
├── provider.tf     (Always provider config)
├── main.tf         (Always main resources)
├── variables.tf    (Always variables)
├── outputs.tf      (Always outputs)
└── terraform.tfvars (Always variable values)
```

Everyone in organization follows same structure

Benefits:

- New team members understand quickly
- Easy to review
- Consistent quality
- Best practices enforced

TERRAFORM LIFECYCLE

Three main steps:

STEP 1: WRITE TERRAFORM CONFIGURATION FILE

How: Write Terraform configuration file with the help of documentation

Documentation: HashiCorp Terraform documentation at terraform.io

What you write:

- Provider details
- Resources to create
- Variables
- Outputs

STEP 2: DRY RUN (PLAN)

Purpose: Before we even execute, we will dry run the file to check if all the code or configuration is correct.

Command:

terraform plan

What it does:

- Validates syntax
- Checks configuration
- Shows what resources will be created
- Shows what will be changed
- Shows what will be destroyed
- No actual changes made

Benefit: We will understand what all resources will be created when we do terraform apply.

STEP 3: APPLY (CREATE RESOURCES)

Command:

terraform apply

What it does: Once we do this, the resources will be created.

Process:

1. Terraform reads configuration
2. Converts to API calls

3. Makes API requests to cloud provider
4. Provider creates resources
5. Terraform updates state file
6. Shows results

TERRAFORM LIFECYCLE DIAGRAM

Step 1: Write Configuration

↓

Step 2: terraform plan (Dry run)

↓

Review planned changes

↓

Step 3: terraform apply (Execute)

↓

Resources created

↓

State file updated

PRACTICAL DEMONSTRATION

STEP 1: INSTALL TERRAFORM

On existing EC2 instance:

Update packages:

```
sudo apt-get update
```

Install dependencies:

```
sudo apt-get install -y gnupg software-properties-common
```

Install Terraform:

```
sudo snap install terraform --classic
```

Verify installation:

terraform --version

Expected output:

Terraform v1.6.0

TERRAFORM FOUR ESSENTIAL COMMANDS

Terraform basically runs on 4 commands:

1. terraform init Initialize Terraform and download providers

2. terraform plan Show what changes will be made

3. terraform apply Create/update resources

4. terraform destroy Delete all resources

STEP 2: UNDERSTAND TERRAFORM INIT

What it does: Terraform init will initialize your Terraform to initialize your provider and stuff.

Process: If we have main.tf file and we have written provider details in that file, and if we do terraform init in the directory where main.tf is present, whatever we wrote in main.tf, that provider configuration will be created in your local and you will be able to communicate with that provider.

Example:

terraform init

What happens:

1. Reads provider configuration
2. Downloads provider plugins
3. Initializes backend
4. Prepares working directory

Output:

Initializing provider plugins...

- Finding hashicorp/aws versions matching "~> 4.16"...

```
- Installing hashicorp/aws v4.67.0...
- Installed hashicorp/aws v4.67.0


Terraform has been successfully initialized!
```

TERRAFORM CONFIGURATION FILE STRUCTURE

What are the actual things that we are going to write in our Terraform file?

BLOCK 1: TERRAFORM BLOCK (REQUIRED)

Purpose: Configure Terraform settings

Content:

hcl

```
terraform {
 required_providers {
  aws = {
   source  = "hashicorp/aws"
   version = "~> 4.16"
  }
 }

 required_version = ">= 1.2.0"
}
```

EXPLANATION:

terraform block: Main configuration block for Terraform

required_providers: Specifies which providers to use

In case of single cloud provider or multiple providers: The required_providers details are static, only the version might change.

Where to get this information: We can get this info from Terraform documentation for AWS provider.

aws = {}: Provider name (can be aws, azurerm, google, etc.)

source = "hashicorp/aws": Where to download provider from

version = "~> 4.16": Which version to use

- ~> means: Use 4.16 or any higher minor version (4.17, 4.18)
- Not major version change (not 5.0)

required_version = ">= 1.2.0": Minimum Terraform version required This is compulsory to write as well.

ADDING MULTIPLE PROVIDERS

Current configuration:

hcl

```
terraform {
 required_providers {
  aws = {
   source  = "hashicorp/aws"
   version = "~> 4.16"
  }
 }
}
```

Future: Adding Azure

hcl

```
terraform {
 required_providers {
  aws = {
   source  = "hashicorp/aws"
   version = "~> 4.16"
  }
  azurerm = {
    source  = "hashicorp/azurerm"
    version = "~> 3.0"
  }
 }
}
```

After adding: You will reinitialize your Terraform state by running terraform init again.

Why reinitialize:

- Download new provider plugins
- Update configuration
- Prepare for new provider

BLOCK 2: PROVIDER BLOCK (OPTIONAL BUT RECOMMENDED)

Purpose: Configure provider-specific settings

Content:

hcl

```
provider "aws" {
  region = "us-west-2"
}
```

region parameter: Specify which AWS region to use

If you don't write region: By default, us-east-1 will be considered.

This is not a mandatory block: We can add any region or just leave it empty (uses default).

Multiple regions:

hcl

```
provider "aws" {
  region = "us-west-2"
  alias  = "west"
}


provider "aws" {
  region = "us-east-1"
  alias  = "east"
}
```

BLOCK 3: RESOURCE BLOCK (MAIN CONTENT)

Purpose: Define what resources to create

This is the difference between each and every Terraform file you are writing.

What varies: Only the resource block varies.

What is static: The first two blocks (terraform and provider) are mostly static.

Content:

hcl

```
resource "aws_instance" "app_server" {
  ami           = "ami-830c94e3"
  instance_type = "t2.micro"

  tags = {
    Name = "Terraform_Demo"
  }
}
```

` ` `

RESOURCE BLOCK EXPLANATION:

resource keyword:

Declares a resource

"aws_instance":

Resource type from AWS provider

"app_server":

Resource name (you choose this)

ami:

Amazon Machine Image ID

instance_type:

Size of instance (t2.micro, t2.small, etc.)

tags:

Metadata for resource

Where to get this information:

This information we will get from HashiCorp Terraform documentation.

Documentation structure:

```
Go to: registry.terraform.io
Select: AWS provider
Find: aws_instance resource
See: All available parameters
```

BLOCK 4: VARIABLES (OPTIONAL)

Purpose: Store reusable values

File: variables.tf or inputs.tf

Why use variables: All the variable details will be inside inputs.tf files in most organizations.

Benefit: If we want to make any changes, we do not have to make changes in the main.tf or the main Terraform file. We will simply make changes in the inputs.tf file.

This is a good way of organizing documents: We simply segregate the variables and put them in different files.

variables.tf example:

hcl

```
variable "instance_type" {
  description = "EC2 instance type"
  type        = string
  default     = "t2.micro"
}


variable "region" {
  description = "AWS region"
  type        = string
  default     = "us-west-2"
}
```

Using variables in main.tf:

hcl

```
resource "aws_instance" "app_server" {
  ami           = "ami-830c94e3"
  instance_type = var.instance_type


  tags = {
   Name = "Terraform_Demo"
 }
}
```

BLOCK 5: OUTPUTS (OPTIONAL)

Purpose: Display information after resources are created

File: outputs.tf

inputs.tf: For inputting **outputs.tf:** For outputting

WHY USE OUTPUTS:

Scenario: You have already created an EC2 instance on AWS and you want to show in the terminal what are the specific characteristics of this EC2 instance that was created.

Solution: You can add those variables to your output.tf and Terraform will understand that you want to show these details once the resources are created.

EXAMPLE:

Requirement: For EC2 instance, you want to get the private IP address once the instance is created.

Explanation: Private IP address is given by AWS. AWS can allocate you dynamically any private IP address.

Solution: You can mention the details in output.tf file so that Terraform will understand that you are expecting it to share the private IP address and Terraform will print the output of private IP address.

outputs.tf example:

hcl

```
output "instance_private_ip" {
 description = "Private IP address of EC2 instance"
 value       = aws_instance.app_server.private_ip
}
```

```
output "instance_public_ip" {
  description = "Public IP address of EC2 instance"
  value       = aws_instance.app_server.public_ip
}

output "instance_id" {
  description = "ID of EC2 instance"
  value       = aws_instance.app_server.id
}
```

After terraform apply:

```
Outputs:

instance_id = "i-1234567890abcdef0"
instance_private_ip = "172.31.10.25"
instance_public_ip = "54.123.45.67"
```

PREREQUISITE: AWS CLI CONFIGURATION

Question:
How is our Terraform able to login to our AWS account?

Answer:

There is a prerequisite for Terraform to work. We need to configure our AWS CLI first, then only our Terraform will be able to authenticate.

INSTALL AWS CLI:

Command:

```
sudo snap install aws-cli --classic
```

Configure AWS CLI:

```
aws configure
```

Provide details:

1. AWS Access Key ID

2. AWS Secret Access Key

3. Default region name (e.g., us-west-2)

4. Default output format (json)

Verify configuration:

```
aws s3 ls
```

If this works, AWS CLI is configured correctly.

STEP 3: CLONE TERRAFORM PROJECT

Clone repository:
```
git clone https://github.com/iam-veeramalla/write_your_first_terraform_project.git
```

Navigate to project:
```
cd write_your_first_terraform_project
cd aws
cd local_state
```

Check files:
```
ls
```

Output:
```
main.tf
```

STEP 4: EXAMINE MAIN.TF FILE

View content:

```
cat main.tf
```

Content:

```hcl
terraform {
  required_providers {
    aws = {
      source  = "hashicorp/aws"
      version = "~> 4.16"
    }
  }

  required_version = ">= 1.2.0"
}

provider "aws" {
  region  = "us-west-2"
}

resource "aws_instance" "app_server" {
  ami           = "ami-830c94e3"
  instance_type = "t2.micro"
  tags = {
    Name = "Terraform_Demo"
  }
}
```

FILE EXPLANATION LINE BY LINE

Lines 1-10: terraform block

hcl

```
terraform {
  required_providers {
    aws = {
      source  = "hashicorp/aws"
      version = "~> 4.16"
    }
  }

  required_version = ">= 1.2.0"
}
```

Purpose: Configure Terraform settings

required_providers:

- Tells Terraform to use AWS provider
- Source: Where to download from (HashiCorp registry)
- Version: Which version to use

required_version:

- Minimum Terraform version
- Ensures compatibility
- Required field

Lines 12-14: provider block

hcl

```
provider "aws" {
  region  = "us-west-2"
}
```

Purpose: Configure AWS provider settings

region:

- Which AWS region to create resources in
- us-west-2 = Oregon
- Optional (defaults to us-east-1)

Lines 16-23: resource block

hcl

```
resource "aws_instance" "app_server" {
  ami           = "ami-830c94e3"
  instance_type = "t2.micro"

  tags = {
    Name = "Terraform_Demo"
  }
}
```

` ` `

Purpose:

Define EC2 instance to create

resource "aws_instance":

- Resource type

- From AWS provider

- Creates EC2 instance

"app_server":

- Resource name (local reference)

- You can choose any name

- Used in Terraform code only

ami:

- Amazon Machine Image ID

- Defines operating system

- Region-specific

instance_type:

- Size of instance

- t2.micro = 1 CPU, 1GB RAM

- Free tier eligible

tags:

- Metadata for resource

- Name tag shows in AWS console

- Used for organization

STEP 5: INITIALIZE TERRAFORM

Command:

```
terraform init
```

Output:

```
Initializing the backend...

Initializing provider plugins...
- Finding hashicorp/aws versions matching "~> 4.16"...
- Installing hashicorp/aws v4.67.0...
- Installed hashicorp/aws v4.67.0 (signed by HashiCorp)

Terraform has been successfully initialized!
```

What happened:

- Downloaded AWS provider plugin
- Initialized working directory
- Created .terraform folder
- Ready to use

STEP 6: PLAN CHANGES

Command:

```
```

terraform plan
```

**Output:**
```
Terraform will perform the following actions:

 # aws_instance.app_server will be created
 + resource "aws_instance" "app_server" {
   + ami   = "ami-830c94e3"
   + instance_type   = "t2.micro"
   + id   = (known after apply)
   + private_ip   = (known after apply)
   + public_ip   = (known after apply)
   + tags   = {
     + "Name" = "Terraform_Demo"
    }
   # ... other attributes
  }

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

**Explanation:**

- + symbol: Resource will be created
- ~ symbol: Resource will be modified
- - symbol: Resource will be destroyed
```

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

- 1 new resource

- 0 modifications

- 0 deletions

STEP 7: APPLY CHANGES

Command:
```
terraform apply
```

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

  Enter a value:
```

Type: yes

Press Enter

Execution output:

```
aws_instance.app_server: Creating...

aws_instance.app_server: Still creating... [10s elapsed]

aws_instance.app_server: Still creating... [20s elapsed]

aws_instance.app_server: Creation complete after 25s [id=i-0abcd1234efgh5678]


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

Result:

Instance has been created in us-west-2 region with the details given in main.tf file.

The instance ID will also be printed to CLI once it is created: i-0abcd1234efgh5678

STEP 8: VERIFY ON AWS CONSOLE

Login to AWS Console:

1. Go to EC2 Dashboard
2. Change region to us-west-2
3. Go to Instances
4. You will see instance named "Terraform_Demo"
5. Instance is running

This is about creation of EC2 instance using Terraform.

ADVANCED CONCEPTS

OUTPUT.TF FILE - USE CASE

Scenario:

There is a user who has executed my script but he doesn't have access to AWS console at all.

User situation:

- When he runs the Terraform script
- He can just see the instance ID
- But he also needs to know:
  - Private IP address
  - Public IP address
  - Key pair name (so he can login)
  - Other details related to instance

Why this situation:

- He is the user who actually requested this instance
- He does not have access to cloud platform
- He used Jenkins pipeline to run Terraform script

Solution:

For that reason, we should create a file called output.tf and provide all the details that Terraform script execution should give as output when main script is executed.

CREATE OUTPUT.TF FILE

Command:

` ` `

```
vim output.tf
```

Content:

hcl

```
output "instance_id" {
  description = "ID of the EC2 instance"
  value       = aws_instance.app_server.id
}

output "instance_private_ip" {
  description = "Private IP address of EC2 instance"
  value       = aws_instance.app_server.private_ip
}

output "instance_public_ip" {
  description = "Public IP address of EC2 instance"
  value       = aws_instance.app_server.public_ip
}

output "instance_public_dns" {
  description = "Public DNS of EC2 instance"
  value       = aws_instance.app_server.public_dns
}

output "key_name" {
  description = "Key pair used"
  value       = aws_instance.app_server.key_name
}
```

Run terraform apply again:

```
terraform apply
```

Output now shows:

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

Outputs:

instance_id = "i-0abcd1234efgh5678"
instance_private_ip = "172.31.10.25"
instance_public_ip = "54.123.45.67"
instance_public_dns = "ec2-54-123-45-67.us-west-2.compute.amazonaws.com"
key_name = "mykey"
```

User can now see all information needed to access the instance!

VARIABLES.TF OR INPUTS.TF FILE

Purpose: Store variable definitions

Why use: We can write the variable-related information in either variable.tf file or inputs.tf file which can be used inside the main.tf file.

variables.tf example:

```
hcl

variable "instance_type" {

  description = "Type of EC2 instance"

  type        = string

  default     = "t2.micro"

}


variable "ami_id" {

  description = "AMI ID for instance"

  type        = string

}


variable "instance_name" {

  description = "Name tag for instance"

  type        = string

  default     = "MyInstance"

}
```

Updated main.tf:

```
hcl

resource "aws_instance" "app_server" {

  ami           = var.ami_id

  instance_type = var.instance_type


  tags = {

    Name = var.instance_name
```

```
 }
}
```

terraform.tfvars (variable values):

```
hcl
ami_id      = "ami-830c94e3"
instance_type = "t2.small"
instance_name = "ProductionServer"
```

Benefits:

- Easy to change values

- Reusable configuration

- Different values for dev/prod

- Clean main.tf

TERRAFORM STATE FILE

WHAT IS TERRAFORM.TFSTATE

When you did terraform apply:

A new file named terraform.tfstate was created in the directory where main.tf was present.

Location:

Same directory as main.tf

Check:

```
ls
```

Output:

```
main.tf  terraform.tfstate  terraform.tfstate.backup
```

IMPORTANCE OF STATE FILE

This is the important file of your entire Terraform.

This is Terraform state file.

What happens if deleted:

If I delete this terraform.tfstate file, Terraform will not understand that it had created EC2 instance for you.

Why:

Terraform is also about tracking the infrastructure.

How Terraform tracks:

Whenever we open this file, we can see that Terraform has written all that it has created till now inside this file for remembering.

STATE FILE EXAMPLE

Open state file:

```
cat terraform.tfstate
```

Content (example):

json

```
{
 "version": 4,
 "terraform_version": "1.6.0",
 "serial": 1,
 "lineage": "abc-123-def",
 "outputs": {},
 "resources": [
  {
   "mode": "managed",
   "type": "aws_instance",
   "name": "app_server",
   "provider": "provider[\"registry.terraform.io/hashicorp/aws\"]",
   "instances": [
    {
     "attributes": {
      "id": "i-0abcd1234efgh5678",
      "ami": "ami-830c94e3",
      "instance_type": "t2.micro",
      "private_ip": "172.31.10.25",
      "public_ip": "54.123.45.67",
      "tags": {
       "Name": "Terraform_Demo"
      }
     }
    }
   ]
  }
 ]
}
```

Information stored:

- Resource IDs
- Current configuration
- IP addresses
- All attributes
- Dependencies

HOW STATE FILE DETECTS CHANGES

Example:

Step 1: Open main.tf

```
vim main.tf

**Step 2: Change instance_type**

hcl

resource "aws_instance" "app_server" {
  ami           = "ami-830c94e3"
  instance_type = "t1.micro"  # Changed from t2.micro

  tags = {
    Name = "Terraform_Demo"
  }
}
```

Save and exit

Step 3: Run terraform plan

```
terraform plan
```

Output:

```
Terraform will perform the following actions:

  # aws_instance.app_server will be updated in-place
  ~ resource "aws_instance" "app_server" {
      id          = "i-0abcd1234efgh5678"
    ~ instance_type = "t2.micro" -> "t1.micro"
      tags        = {
          "Name" = "Terraform_Demo"
      }
  }

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

How Terraform knew:

Terraform got to know about this change because of this state file.

Process:

1. Terraform reads current configuration (main.tf)
2. Terraform reads state file (what was created before)
3. Compares both
4. Finds difference: t2.micro → t1.micro
5. Plans to update resource

WHAT STATE FILE CONTAINS

This file has:

- Sensitive information (passwords, keys)
- Non-sensitive information (names, IDs)
- Information about VPC

- Information about EC2 instances

- All resource details

If organization is maintaining entire AWS account using Terraform:

State file will have information right from:

- CloudWatch alarms

- EC2 instances

- VPCs

- KMS keys

- S3 buckets

- RDS databases

- IAM users

- Everything created by Terraform

THE STATE FILE PROBLEM

Current situation:

Right now I have executed terraform apply from my laptop and then this terraform.tfstate file is created on my laptop and with open permissions, so it can be read by all the users.

Initial thought:

We might think this is not a big problem. I can simply use Linux commands and change the permissions of this file.

Command:

```
```

chmod 600 terraform.tfstate

```

**Yes, we can do it, but there's a bigger problem:**

---

**THE SHARING PROBLEM**

**Scenario:**

Let's say:

- We have cloned this repository
- The person who created this repo and me are working in the same organization
- We both are using this repository to create infrastructure on AWS

**What happens:**

**Person 1 executes terraform script:**

- State file created on Person 1's local machine
- Contains infrastructure details

**Person 2 executes terraform script:**

- State file created on Person 2's local machine
- Contains same infrastructure details

**Problems:**
```

Problem 1:

This state file cannot be uploaded to Git repository because it has a lot of sensitive information.

Problem 2:

We cannot merge both state files.

Problem 3:

Two people have different views of infrastructure.

Problem 4:

Conflicts when both try to update same resource.

THE SOLUTION: REMOTE BACKEND

To solve this issue:

We have to put the state file in one centralized location.

Rule 1:

You should store your state files remotely, not on your local machine.

Rule 2:

It is not a good idea to store the state file in source control like GitHub either, as I mentioned earlier.

Rule 3:

We should store the state files remotely in some remote backend.

What is remote backend:

Centralized storage for state files

- Amazon S3 (for AWS)

- Azure Blob Storage (for Azure)

- Google Cloud Storage (for GCP)

- Terraform Cloud

- HashiCorp Consul

STATE FILE SECURITY RULES

Rule 1: Do not manipulate state files locally

If we try to update the state file manually, then Terraform will get corrupted.

Rule 2: Do not modify state file on your own

Always give only read permission to state file. Only Terraform should be able to update it.

Commands that modify state (use carefully):

- terraform state mv

- terraform state rm

- terraform import

- terraform refresh (deprecated)

Never:

- Edit JSON directly

- Delete entries manually

- Modify structure

IDEAL TERRAFORM SETUP

Setup description:

Step 1: DevOps engineers write Terraform scripts

Step 2: Store scripts in Git repository

Step 3: Users in organization want to use these scripts

As DevOps engineer decision:

I did not grant them access to AWS infrastructure. Apart from DevOps team, nobody will log into AWS for some reason. Nobody else has access to create resources on AWS.

Step 4: If they want to create resources

They can use the Jenkins pipeline which I have created.

Jenkins pipeline workflow:

1. Jenkins will watch for Terraform resources in GitHub

2. Jenkins will pull .tf files from GitHub

3. Jenkins will execute terraform apply

4. Resources created on AWS

5. Person using outputs.tf will get the information of resources created

Let's say this is the process.

ADDITION: REMOTE STATE IN S3

What I have added to this process:

The state file is stored in Amazon S3.

Why:

It is a good practice to store the state files in Amazon S3 rather than:

- On GitHub
- On local machine
- In Jenkins

If we are using AWS:

S3 bucket acts like a centralized location.

How it works:

If someone executes the Terraform scripts, Terraform will automatically update the state file in centralized location (S3 bucket).

CHALLENGE: PARALLEL EXECUTION

Problem scenario:

Two people are trying to execute Terraform scripts parallely.

Example:

- Person 1: "Update EC2 instance to 2 CPUs"
- Person 2: "Update EC2 instance to 4 CPUs"
- Both running at same time

Question:

Which instruction will Terraform take?

Terraform should not allow parallel execution because:

- Conflicting changes
- State file corruption
- Unpredictable results
- Resource inconsistency

SOLUTION: STATE LOCKING WITH DYNAMODB

To avoid that problem:

You will integrate your backend with DynamoDB.

What DynamoDB does:

DynamoDB will be used for locking the Terraform state file.

How locking works:

Person 1 starts terraform apply:

1. Terraform locks state file in DynamoDB

2. Person 1's changes being applied

Person 2 tries terraform apply:

1. Terraform checks lock

2. Finds state file is locked

3. Terraform says: "State file is locked. Some other user is actually using the Terraform script or some other user is performing infrastructure configuration on your AWS resources. Wait until the configuration is done."

Person 1 completes:

1. Terraform completes changes

2. Unlocks state file in DynamoDB

Person 2 can now proceed:

1. Terraform sees state file is unlocked

2. Locks it

3. Executes Person 2's changes

Benefit:

No conflicts, sequential execution, safe updates.

IDEAL TERRAFORM CONFIGURATION SUMMARY

Best practices:

1. Always put Terraform scripts in GitHub or any version control system

2. Terraform state files should go in remote backends

What is remote backend:

Remote storage services like:

- Amazon S3 bucket
- Azure Storage Container
- Google Cloud Storage
- Terraform Cloud

3. Store state files there

4. Integrate with proper locking solutions

In case of AWS:

The locking solution is DynamoDB.

Once you integrate remote backend with DynamoDB:

Terraform will not allow parallel execution of Terraform scripts.

LOCAL STATE vs REMOTE STATE

Local state (what we did earlier):

- State file on local machine
- Good for learning
- Not good for teams
- Not secure
- Not shared

Remote state (production):

- State file in S3
- Good for teams
- Secure
- Shared
- Locked with DynamoDB
- Best practice

In real-time scenarios or production scenarios:

That is not how you deal with things. You always configure a remote backend.

ENVIRONMENT ISOLATION

In an organization, there are many types of environments:

- UAT (User Acceptance Testing)
- Development
- Staging

- Production

Best practice:

It is always a good practice to isolate Terraform scripts for each and every environment.

How:

We will have separate Terraform files for each environment.

Structure:

```
terraform/
 ├── dev/
│   ├── main.tf
│   ├── variables.tf
│  └── terraform.tfstate (in S3)
 ├── staging/
│   ├── main.tf
│   ├── variables.tf
│  └── terraform.tfstate (in S3)
└── production/
    ├── main.tf
    ├── variables.tf
   └── terraform.tfstate (in S3)
```

Why isolate:

Isolate and organize the state files to reduce blast radius.

What is blast radius:

If something goes wrong, damage is limited to one environment.

Example:

- Bug in dev environment script

- Only dev affected

- Staging and production safe

REMOTE BACKEND CONFIGURATION

STEP 1: NAVIGATE TO REMOTE STATE FOLDER

Command:

```
cd ..

cd remote_state

ls
```

Files present:

```
main.tf

outputs.tf

variables.tf
```

STEP 2: EXAMINE MAIN.TF FOR REMOTE BACKEND

Content:

hcl

```
#
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

# CREATE AN S3 BUCKET AND DYNAMODB TABLE TO USE AS A TERRAFORM BACKEND

#
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~


# ---------------------------------------------------------------------------------------------------------
----------

# REQUIRE A SPECIFIC TERRAFORM VERSION OR HIGHER

# This module has been updated with 0.12 syntax

# This module is forked from https://github.com/gruntwork-io/intro-to-
terraform/tree/master/s3-backend

# ---------------------------------------------------------------------------------------------------------
----------


terraform {

  required_version = ">= 0.12"

}


# ------------------------------------------------------------------------------

# CONFIGURE OUR AWS CONNECTION

# ------------------------------------------------------------------------------


provider "aws" {}
```

```
# ------------------------------------------------------------------------------
# CREATE THE S3 BUCKET
# ------------------------------------------------------------------------------

data "aws_caller_identity" "current" {}

locals {
  account_id = data.aws_caller_identity.current.account_id
}

resource "aws_s3_bucket" "terraform_state" {
  bucket = "${local.account_id}-terraform-states"

  versioning {
    enabled = true
  }

  server_side_encryption_configuration {
    rule {
      apply_server_side_encryption_by_default {
        sse_algorithm = "AES256"
      }
    }
  }
}

# ------------------------------------------------------------------------------
# CREATE THE DYNAMODB TABLE
# ------------------------------------------------------------------------------

resource "aws_dynamodb_table" "terraform_lock" {
  name         = "terraform-lock"
  billing_mode = "PAY_PER_REQUEST"
  hash_key     = "LockID"

  attribute {
    name = "LockID"
    type = "S"
  }
}
```

FILE EXPLANATION

Purpose: This script creates the infrastructure needed for remote backend:

1. S3 bucket for state storage
2. DynamoDB table for state locking

Section 1: Terraform version

hcl

```
terraform {
  required_version = ">= 0.12"
}
```

Requires Terraform 0.12 or higher

Section 2: AWS provider

hcl

```
provider "aws" {}
```

Uses default AWS configuration from aws configure

Section 3: Get AWS account ID

hcl

```
data "aws_caller_identity" "current" {}


locals {
  account_id = data.aws_caller_identity.current.account_id
}
```

data source: Queries AWS for current account information

locals: Creates local variable with account ID

Why: Use account ID in S3 bucket name for global uniqueness

Section 4: Create S3 bucket

hcl

```
resource "aws_s3_bucket" "terraform_state" {
  bucket = "${local.account_id}-terraform-states"


  versioning {
    enabled = true
  }


  server_side_encryption_configuration {
    rule {
      apply_server_side_encryption_by_default {
        sse_algorithm = "AES256"
      }
  }
 }
}
```

bucket name: Uses account ID to make globally unique Example: 123456789012-terraform-states

versioning: Enabled so we can see full revision history of state files

- Keep old versions
- Recover from mistakes
- Track changes

encryption: Enable server-side encryption by default

- Secures state file
- AES256 encryption
- Protects sensitive data

Section 5: Create DynamoDB table

```hcl
resource "aws_dynamodb_table" "terraform_lock" {
 name        = "terraform-lock"
 billing_mode = "PAY_PER_REQUEST"
 hash_key     = "LockID"

 attribute {
  name = "LockID"
  type = "S"
 }
}
```

name:

Table name: terraform-lock

billing_mode:

PAY_PER_REQUEST means pay only for reads/writes

hash_key:

Primary key for table: LockID

attribute:

- LockID: String type

- Used to identify locks

Purpose:

This table manages locks when multiple users try to run Terraform

---

STEP 3: CREATE BACKEND INFRASTRUCTURE

Initialize:

```
terraform init
```

Plan:

```
terraform plan
```

` ` `

Apply:

` ` `

```
terraform apply
```

Type: yes

Result:

- S3 bucket created for state storage
- DynamoDB table created for locking

STEP 4: CONFIGURE BACKEND IN MAIN PROJECT

Go back to your main project:

Edit main.tf, add backend configuration:

hcl

```
terraform {
 required_providers {
  aws = {
   source  = "hashicorp/aws"
   version = "~> 4.16"
  }
 }

 required_version = ">= 1.2.0"

 backend "s3" {
  bucket       = "123456789012-terraform-states"
  key        = "dev/terraform.tfstate"
  region       = "us-west-2"
```

```
    dynamodb_table = "terraform-lock"
    encrypt        = true
  }
}

provider "aws" {
  region = "us-west-2"
}

resource "aws_instance" "app_server" {
  ami           = "ami-830c94e3"
  instance_type = "t2.micro"

  tags = {
    Name = "Terraform_Demo"
  }
}
```

Re-initialize:

```
terraform init
```

Output:

```
Initializing the backend...
```

```
Successfully configured the backend "s3"!
```

State file now in S3!

TERRAFORM MODULES

WHAT ARE TERRAFORM MODULES

Definition:

Whenever we have reusable code like creation of S3 bucket, AWS DynamoDB table, etc., we can put this script in a module.

Concept:

Let's say this happens to be reusable code in your organization.

What you do:

Put this script in GitHub.

What others do:

Others can use this repository as a module inside other Terraform scripts or inside other Terraform files.

HOW MODULES WORK

You will reference these things as modules in Terraform.

Once we reference them as modules:

Before execution of main scripts, Terraform will start executing the modules.

Example:

Scenario:

Someone uses our current script (S3 + DynamoDB) as a module.

What happens by default:

They will have Terraform configured with:

- S3 remote backend

- DynamoDB enabled

Benefit:

There is a lot of effort that has gone away. Modules are a way of writing reusable components.

---

MODULE EXAMPLE

Create module (s3-backend module):

Directory structure:

```
modules/
└── s3-backend/
    ├── main.tf
    ├── variables.tf
    └── outputs.tf
```

modules/s3-backend/main.tf:

hcl

```hcl
resource "aws_s3_bucket" "terraform_state" {
  bucket = var.bucket_name

  versioning {
    enabled = true
  }

  server_side_encryption_configuration {
    rule {
      apply_server_side_encryption_by_default {
        sse_algorithm = "AES256"
      }
    }
  }
}

resource "aws_dynamodb_table" "terraform_lock" {
  name         = var.table_name
  billing_mode = "PAY_PER_REQUEST"
  hash_key     = "LockID"
  attribute {
    name = "LockID"
    type = "S"
  }
}
```

Use module in main project:

main.tf:

```hcl
module "backend" {
  source      = "./modules/s3-backend"
  bucket_name = "my-terraform-state"
  table_name  = "my-terraform-lock"
}
```

Run:

```
terraform init
terraform apply
```

Result:

Module creates S3 bucket and DynamoDB table automatically.

DISADVANTAGES OF TERRAFORM

DISADVANTAGE 1: STATE FILE AS SINGLE SOURCE OF TRUTH

Issue:

State file is a single source of truth.

Problem:

By chance, if this file gets corrupted or you have misconfigured it, that means you have messed up with your infrastructure and Terraform.

Consequences:

- Terraform doesn't know what exists
- Cannot update resources
- Cannot destroy resources
- Manual recovery needed

Example:

```
State file corrupted

Terraform thinks resources don't exist

But they exist on AWS

Terraform tries to create again

Conflicts occur
```

Mitigation:

- Use remote backend with versioning

- Regular backups

- State locking

- Don't edit manually

DISADVANTAGE 2: MANUAL CHANGES NOT DETECTED

Issue:

Manual changes to the cloud provider cannot be identified and auto-corrected.

Scenario:

Step 1:

Somebody wants to change AWS infrastructure and has access to your AWS.

Step 2:

They manually modified some things on AWS.

Example:

We wrote Terraform script to create EC2 instance with t2.micro.

Someone manually updated this EC2 instance to t2.small on AWS console.

Problem:

Terraform is not bidirectional.

What this means:

Direction 1 (Works):

If you make changes in script and execute it, Terraform will know it and apply changes.

Direction 2 (Doesn't work):

If you manually make changes to the resources on AWS, then Terraform will not know this.

What happens next:

Next time you try to update something using Terraform:

Terraform will say:

"Whatever is there in the state file is not same as it is on the cloud provider. Maybe somebody accidentally modified it. Now I cannot update it because there is a difference in the resource."

Terraform detects drift:

```
terraform plan

Output:
Warning: Resource drift detected

  # aws_instance.app_server has been changed
  ~ instance_type = "t2.micro" -> "t2.small"
```

Solutions:

Option 1: terraform refresh (deprecated)

Update state file with current AWS state

Option 2: terraform apply -refresh-only

Refresh state without making changes

Option 3: Manually update state

Use terraform state commands (risky)

Option 4: Revert manual change

Change AWS back to match Terraform

Best practice:

Prevent manual changes. Use Terraform only. Enable CloudTrail to track manual changes.

This is one of the disadvantages of Terraform: It is not bidirectional.

Terraform tracks:

Code → Cloud (Yes)

Terraform doesn't track:

Cloud → Code (No)

DISADVANTAGE 3: NOT GITOPS FRIENDLY

Issue:

Terraform does not play well with Flux or Argo CD.

What is GitOps:

Git as single source of truth. Any changes in Git automatically applied to infrastructure.

GitOps tools:

- Flux

- Argo CD

Problem:

Scenario:

We have Terraform script on AWS. If we make changes to the resource on AWS cloud itself, now the information is corrupted. Git is not the single source of truth anymore.

Why not GitOps friendly:

- Terraform requires manual execution

- State file separate from Git

- Drift not automatically corrected

- Manual changes break Git truth

GitOps workflow (ideal):

```
Change in Git → Automatic deployment → Infrastructure updated
```

Terraform reality:

```
Change in Git → Manual terraform apply → Infrastructure updated
```

Workarounds:

- Use Atlantis (Terraform automation for PR)
- Use Terraform Cloud
- Custom CI/CD pipelines

DISADVANTAGE 4: COMPLEXITY

Issue:

Terraform can become very complex and difficult to manage.

Reasons:

1. Large state files:

- Thousands of resources
- Slow operations
- Hard to navigate

2. Module dependencies:

- Complex relationships
- Circular dependencies
- Hard to debug

3. Provider limitations:

- Different provider behaviors
- Version compatibility
- Breaking changes

4. State management:

- Locking issues
- Corruption risks
- Migration challenges

5. Learning curve for advanced features:

- Workspaces
- Dynamic blocks
- For_each loops
- Complex expressions

Mitigation:

- Break into smaller projects
- Use modules wisely
- Good documentation
- Team training

DISADVANTAGE 5: TOOL OVERLAP WITH ANSIBLE

Issue:

Terraform is trying to position as a configuration management tool as well.

What this means:

Nowadays, Ansible and Terraform are trying to cross paths with each other.

Example:

People are also using Ansible to create infrastructure.

Problem:

Tools should be used accordingly.

Correct usage:

Terraform:

Should be used for infrastructure creation

- Create EC2 instances
- Create VPCs
- Create S3 buckets
- Provision resources

Ansible:

Should be used for configuration management only

- Install software
- Configure services

- Manage files
- Update configurations

Why this overlap is a problem:

Terraform for configuration:
- Not designed for it
- Less flexible than Ansible
- Harder to maintain
- Not idempotent for config

Ansible for infrastructure:
- Can create resources
- But less efficient than Terraform
- No state management
- Harder to track

Best practice:
Use both together:
1. Terraform creates infrastructure
2. Ansible configures servers

Example workflow:
```
terraform apply (creates 10 EC2 instances)
   ↓
```

ansible-playbook configure.yml (installs software on all 10)

Clear separation of concerns:

- Terraform: What to create
- Ansible: How to configure

SUMMARY

Why Terraform:

1. Multi-cloud support
2. Infrastructure visibility via state
3. Automate changes
4. Collaboration via Git
5. Standardized configurations

Terraform lifecycle:

1. Write configuration
2. terraform plan (dry run)
3. terraform apply (create resources)

Four essential commands:

- terraform init
- terraform plan
- terraform apply
- terraform destroy

State file:

- Tracks infrastructure
- Single source of truth
- Must be protected
- Store remotely (S3)
- Lock with DynamoDB

Ideal setup:

- Scripts in Git
- State in S3
- Lock with DynamoDB
- Execute via CI/CD (Jenkins)
- Outputs for users

Environment isolation:

- Separate folders for dev/staging/prod
- Reduce blast radius
- Independent state files

Modules:

- Reusable components
- Share via GitHub or Terraform Registry
- Reduce duplication

Disadvantages:

1. State file corruption risk
2. Manual changes not tracked
3. Not GitOps friendly
4. Can become complex
5. Tool overlap with Ansible

Best practice:

- Terraform for infrastructure
- Ansible for configuration
- Use both together

This is a comprehensive guide to Terraform for DevOps engineers.