DEVOPSDIVE
A WOMAN'S DIVE INTO DEVOPS EXCELLENCE

TERRAFORM & INFRASTRUCTURE AS CODE

Terraform = make a planet like Earth.

LEARN TO PROVISION, MANAGE, AND SCALE CLOUD INFRASTRUCTURE WITH CONFIDENCE

Multi-Cloud ***Declarative*** ***Version Control***

WHY LEARN TERRAFORM?

Manual Infrastructure : Slow, Error-prone, Expensive

Terraform : Fast, Reliable, Career Boost!

Infrastructure as Code

Define and manage infrastructure using declarative configuration files

Works Everywhere

AWS, Azure, GCP - same commands, different clouds

Save Time

No more clicking in consoles - automate everything

Version Control

Track all infrastructure changes

Collaboration

Team-friendly with remote state and workspace management

Scalability

Manage from simple VMs to complex multi-cloud architectures

TERRAFORM VS ANSIBLE

Aspect	Terraform	Ansible
Primary Purpose	Infrastructure Provisioning	Configuration Management
Approach	Declarative (What you want)	Procedural (How to do it)
Best For	Creating cloud resources	Configuring existing systems

Use Both Together!

Terraform creates infrastructure → Ansible configures applicati

IaC Evolution Timeline

2011
AWS CloudFormation

2014
Terraform Launch

Present
Multi-Cloud Era

Key Point:

AWS CloudFormation came first (2011) but was limited to AWS only. Terraform (2014) introduced the multi-cloud approach!

Two Approaches to IaC

IMPERATIVE

Defines the specific commands needed to achieve the desired configuration.

Commands must be executed in correct order

VS

DECLARATIVE

Defines the desired end state of infrastructure.

Tool figures out how to achieve it.

Terraform uses Declarative Approach!

sample Terraform file (main.tf)

```
terraform {
  required_providers {
    aws = {
      source  = "hashicorp/aws"
      version = "~> 5.92"
    }
  }
  required_version = ">= 1.2"
}
provider "aws" {
  region = "us-east-1"
}
#Sample EC2 instance (Ubuntu)
resource "aws_instance" "app_server" {
  ami           = "ami-0c55b159cbfafe1f0"
  instance_type = "t2.micro"
  tags = {
    Name = "learn-terraform"
  }
}
```

Terraform Introduction

- First Released: July 2014 (supported AWS & Digital Ocean)
- Created by: Mitchell Hashimoto
- Company: HashiCorp
- Written in: Go Language
- Language Used: HashiCorp Configuration Language (HCL)
- HCL Features: Similar to JSON, but easy and human readable

Terraform is a tool for Infrastructure as Code (IaC) that lets you create, update, and manage cloud resources (like VMs, networks, and storage) using code. It tracks resource state, supports multiple providers, and makes infrastructure automated, repeatable, and reusable.

HashiCorp Configuration Language (HCL)

Basic Syntax Structure

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

# Resource
resource "aws_instance" "example" {
  ami           = "ami-12345678"
  instance_type = "t2.micro"
  tags = { Name = "MyEC2Instance" }
}

# Variable
variable "instance_count" {
  type    = number
  default = 1
}

# Output
output "instance_id" {
  value = aws_instance.example.id
}
```

Core building blocks

- ✓ provider → Cloud/service
- ✓ resource → Infra to create
- ✓ variable → Input values
- ✓ output → Show results

File Types

- ✓ .tf → Configuration files
- ✓ .tfvars → Variable values
- ✓ .tfstate → State files
- ✓ .terraform.lock.hcl → Dependency lock

HCL Features

- ✓ Human-readable syntax
- ✓ JSON compatible
- ✓ Built-in functions
- ✓ Conditional expressions
- ✓ String interpolation

CONNECTING TERRAFORM WITH AWS

AWS CREDENTIALS SETUP

AWS Provider Configuration

```
terraform {
  required_providers {
    aws = {
      source  = "hashicorp/aws"
      version = "~> 5.0"
    }
  }
}
```

AWS CLI

```
aws configure
# Enter Access Key ID
# Enter Secret Access Key
# Enter Region
```

Environment Variables

```
export    AWS_ACCESS_KEY_ID="your-key"
export    AWS_SECRET_ACCESS_KEY="your-secret"
export    AWS_DEFAULT_REGION="us-east-1"
```

IAM Roles (Production Recommended)

```
provider "aws" {
   assume_role {
        role_arn = "arn:aws:iam::123456789012:role/TerraformRole"
}}
```

Terraform Commands - Complete Reference

1. Setup & Version

terraform -help # List all commands
terraform -version # Show Terraform version

2. Initialize Project

terraform init # Download provider plugins
terraform init -upgrade # Upgrade providers/modules

3. Code Validation & Formatting

terraform validate # Validate .tf files
terraform fmt # Format files
terraform fmt -recursive # Format entire project

4. Planning & Applying

terraform plan # Preview changes
terraform plan -out=tfplan # Save plan to file
terraform apply # Apply changes
terraform apply tfplan # Apply saved plan
terraform apply -auto-approve # Skip confirmation
terraform apply --target=aws_instance.demo_ec2 # Apply specific resource

5. Outputs

terraform output # Show all outputs
terraform output ec2_public_ip # Show one output

6. Destroying

terraform destroy # Destroy infra
terraform destroy -auto-approve # Skip confirmation
terraform destroy --target=aws_s3_bucket.demo # Destroy only one resource

7. State Management

terraform state list # Show all tracked resources
terraform state show aws_instance.demo # Show one resource
terraform state pull # Output raw state (JSON)
terraform state push terraform.tfstate # Push local state to remote
terraform state rm aws_instance.demo # Remove resource from state
terraform state mv aws_instance.old aws_instance.new # Rename/move in state

8. Workspaces (Multi-Env)

terraform workspace list # Show all workspaces
terraform workspace new dev # Create new workspace
terraform workspace select dev # Switch workspace
terraform workspace delete dev # Delete workspace

9. Variables

terraform plan -var="instance_type=t2.micro"
terraform apply -var-file=dev.tfvars

Advanced Commands & Best Practices

10. Lifecycle

terraform taint aws_instance.demo # Mark for recreation
terraform untaint aws_instance.demo # Cancel taint
terraform refresh # Refresh state from provider

11. Modules

terraform get # Download modules
terraform init -upgrade # Upgrade modules

12. Lock & State Backend

terraform init # Configure backend

13. Debugging

export TF_LOG=DEBUG # Enable debug logs
export TF_LOG_PATH=logs.txt # Save logs to file
unset TF_LOG # Disable logging

14. Import & Inspection

terraform import aws_instance.web i-1234567890abcdef0
terraform show # Show current state
terraform graph # Generate dependency graph
terraform console # Interactive console

15. Advanced Options

terraform plan -parallelism=10 # Set concurrency
terraform apply -lock-timeout=5m # Lock timeout
terraform force-unlock LOCK_ID # Force unlock state

Tips for Production

- Always use remote state with state locking
- Implement proper tagging strategy
- Use modules for reusability
- Enable debug logging for troubleshooting
- Regular state backups and validation

TERRAFORM RESOURCES

Resources are the most important element in Terraform. They represent actual infrastructure objects that Terraform creates, updates, or deletes. Examples include VMs, networks, storage, DNS records, databases, etc.

Resource Lifecycle

- **create_before_destroy** - Create new before destroying old
- **prevent_destroy** - Prevent accidental deletion
- **ignore_changes** - Ignore changes to specified attributes

Resource Syntax

```
resource "TYPE" "NAME" {
  arg1 = value1
  arg2 = value2

  nested_block {
    setting = value
  }
}
```

Reference Format

```
# Reference other resources
<RESOURCE_TYPE>.<RESOURCE_NAME>.<ATTRIBUTE>

# Examples
aws_instance.web_server.instance_type
```

- **RESOURCE_TYPE** → Type of resource (aws_instance)
- **RESOURCE_NAME** → Name you gave (web_server)
- **ATTRIBUTE** → Property you want (id, instance_type)

Resource Example

```
resource "aws_instance" "web_server" {
  ami           = "ami-12345678"
  instance_type = "t2.micro"

  tags = {
    Name = "WebServer"
  }
}
```

TERRAFORM VARIABLES

Variables are inputs to your Terraform code that make your infrastructure flexible and reusable. They allow you to avoid hardcoding values.

Ways to Provide Variable Values

- Command line: -var(terraform apply -var="instance_type=t2.small")
- Files: .tfvars (e.g., dev.tfvars)
- Environment: TF_VAR_ (export TF_VAR_instance_type="t2.medium")
- Default value → In variable block

Variable Types

- **string -** Text values
- **number -** Numeric values
- **bool -** true/false
- **list -** Array of values
- **map -** Key-value pairs
- **object -** Complex structures

Variable Syntax

```
variable "<NAME>" {
  description = "Description of the variable"
  type        = string | number | bool | list | map
  default     = <value>   # optional
}
```

Variable Example

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

resource "aws_instance" "web_server" {
  ami           = "ami-12345678"
  instance_type = var.instance_type
}
```

TERRAFORM OUTPUTS

Outputs allow you to display or export values from your Terraform configuration. They are useful for showing resource attributes or passing values to other configurations/modules.

Why Use Outputs?

- Share data between modules
- Display important information after deployment
- Use in remote state data sources
- Integration with external systems

Key Points

- Access resource attributes using TYPE.NAME.ATTRIBUTE
- Can be used in other modules
- sensitive = true hides secret values

Output Syntax

```
output "<NAME>" {
  value       = <expression>
  description = "Optional description"
  sensitive   = true | false   # optional
}
```

Output Example

```
resource "aws_instance" "web_server" {
  ami           = "ami-12345678"
  instance_type = "t2.micro"
}

output "server_id" {
  value       = aws_instance.web_server.id
  description = "ID of the EC2 instance"
}

output "server_type" {
  value = aws_instance.web_server.instance_type
}
```

TERRAFORM PROVISIONERS

Provisioners allow Terraform to execute scripts or commands on a resource after it is created. They are used for bootstrapping, configuration, or initial setup.

When They Run

- Creation-time (default)
- Destroy-time (cleanup)
- Can fail gracefully or stop deployment

⚠️ Important: Provisioners are a last resort! Use cloud-init, user data, or configuration management tools when possible.

Types of Provisioners

- **Local-exec:** Runs commands locally
- **Remote-exec**: Runs commands on remote resource
- **File**: Copies files to remote resource

Terraform Provisioners Types

#local-exec

```
provisioner "local-exec" {
  command = "echo Hello World > hello.txt"
}
```

#remote-exec

```
provisioner "remote-exec" {
  inline = [
    "sudo apt-get update",
    "sudo apt-get install -y nginx"
  ]
}
```

#file

```
provisioner "file" {
  source      = "app.conf"
  destination = "/etc/app/app.conf"
}
```

Syntax (remote-exec example)

```
resource "aws_instance" "web_server" {
  ami           = "ami-12345678"
  instance_type = "t2.micro"

  provisioner "remote-exec" {
    inline = [
      "sudo apt-get update",
      "sudo apt-get install -y nginx"
    ]

    connection {
      type     = "ssh"
      user     = "ubuntu"
      private_key = file("~/.ssh/id_rsa")
      host     = self.public_ip
    }
  }
}
```

TERRAFORM MODULES

Modules are reusable Terraform configurations.They allow you to organize, reuse, and share infrastructure code across projects.

Why Use Modules?

- Reusability: Write once, use many times
- Organization: Group related resources
- Abstraction: Hide complexity
- Sharing: Team collaboration

Module Sources

- **Local paths**
- **Terraform Registry**
- **Git repositories**
- **HTTP URLs**
- **S3 buckets**

Types of Modules

- **Root Module** → The main configuration you run
- **Child Module** → Reusable module called by the root or other modules
- **Registry Module** → Prebuilt modules from Terraform Registry

Memory Tip

Modules = Reusable building blocks

- **Root module** → Main project
- **Child module** → Reusable code
- **Registry module** → Prebuilt shared modules

Syntax (Calling a Module)

```
module "vpc" {
  source = "./modules/vpc"
  cidr_block = "10.0.0.0/16"
  public_subnets = ["10.0.1.0/24", "10.0.2.0/24"]
}
```

Outputs from Modules

```
output "vpc_id" {
  value = module.vpc.vpc_id
}
```

TERRAFORM
STATE MANAGEMENT

Terraform keeps track of resources it creates, updates, or deletes in a state file (terraform.tfstate).The state ensures Terraform knows what exists in your infrastructure.

State is Terraform's way of keeping track of your infrastructure - it's the source of truth!

State Storage

- **Local State:**terraform.tfstate file
- **Good for testing**
- **Not suitable for teams**

Remote State

- Backend Options:S3 + DynamoDB
- Terraform Cloud
- Azure Storage
- Google Cloud Storage

Best Practices

- Use remote state for team collaboration
- Enable state locking
- Encrypt state at rest and in transit
- Regular state backups

State Commands

 # State Commands
terraform state list # List resources in state
terraform state show aws_instance.web # Show specific resource
terraform state mv old_name new_name # Rename resource
terraform state rm aws_instance.old # Remove from state
terraform import aws_instance.web i-123 # Import existing resource

TERRAFORM WORKSPACES

Workspaces allow Terraform to manage multiple environments (e.g., dev, test, prod) using a single configuration.

What are Workspaces?

Workspaces are named containers for Terraform state. Each workspace has its own state file, allowing you to manage different environments with the same code.

Default Workspace

- Every Terraform config starts with "default"
- Cannot be deleted
- Good for single environment

Multiple Workspaces

- Separate state per workspace
- Same configuration, different resources
- Perfect for dev/staging/prod

Considerations:

- Each workspace needs its own backend configuration
- Variables might need different values per workspace
- Not suitable for completely different infrastructures

Using Workspace in Configuration

```
# Using Workspace in Configuration
locals {
  environment = terraform.workspace

  instance_counts = {
    default = 1
    dev     = 1
    staging = 2
    prod    = 5
  }
}

resource "aws_instance" "app" {
  count         = local.instance_counts[local.environment]
  ami           = "ami-12345"
  instance_type = local.environment == "prod" ? "t3.large" : "t2.micro"

  tags = {
    Name        = "${local.environment}-app-${count.index + 1}"
    Environment = local.environment
  }
}
```

CLOUDFORMATION

AWS

ARM TEMPLATE

Azure

DEPLOYMENT MANAGER

GCP

HEAT

OpenStack

CLOUD PROVIDER SUPPORT

TERRAFORM

🎯 Single tool for ALL cloud providers!
100+ providers supported including AWS, Azure, GCP, Kubernetes, Docker, GitHub, and more!

TERRAFORM USE CASES

Multi-Cloud

- *Deploy resources across multiple cloud providers*
- *Avoid vendor lock-in*
- *Use best services from different clouds*

Hybrid Cloud

- *Connect on-premises with cloud resources*
- *Consistent tooling across environments*
- *Gradual cloud migration strategies*

Infrastructure Management

- *Version controlled infrastructure*
- *Automated deployments*
- *Disaster recovery planning*