# Terraform

Infrastructure as a Code

## Introduction

❖ IAAC | Automate Infrastructure

❖ Define Infrastructure State

❖ Ansible, puppet or chef automates mostly OS related tasks.

➢ Defines machines state

❖ Terraform automates infra itself

➢ Like AWS, GCP, Azure, digital ocean etc

❖ Terraform works with automation software’s like ansible after infra is setup and ready.

❖ No Programming, its own syntax similar to JSON.

Everything Needs Automation

Infrastructure automation centralized.

## Installation

Download Terraform binary from its website

❖ Linux

❖ Mac

❖ Windows

Store binary in exported PATH

e:g: Linux => /usr/local/bin

## Launch ec2 instance

❖ AWS Account

❖ IAM User with access keys

❖ Terraform file to launch instance

❖ Run terraform apply

## Exercise

➔ Write instance.tf file

➔ Launch instance

➔ Make some changes to instance.tf file

➔ Apply changes

provider "aws" {

region = "us-east-1"

}

resource "aws\_instance" "intro" {

ami = "ami-0e8a34246278c21e4"

instance\_type = "t2.micro"

availability\_zone = "us-east-1a"

key\_name = "terraform-key"

vpc\_security\_group\_ids = ["sg-002ab56fbcd0c4a86"]

tags = {

Name = "terraform-instance"

project = "terraform.tf"

}

}

terraform –help

aws config

aws configure

504 mkdir terraform-scrits

505 cd terraform-scrits/

506 mkdir exercise1

507 ls

508 cd exercise1/

509 ls

510 terraform init

511 ls -a

512 terraform validate

513 terraform validate

514 terraform validate

515 terraform fmt

516 cat terraform.tf

517 cat first\_instances.tf

518 terraform plan

519 terraform apply

520 terraform validate

521 terraform plan

522 ls

523 cat first\_instances.tf

524 terraform fmt

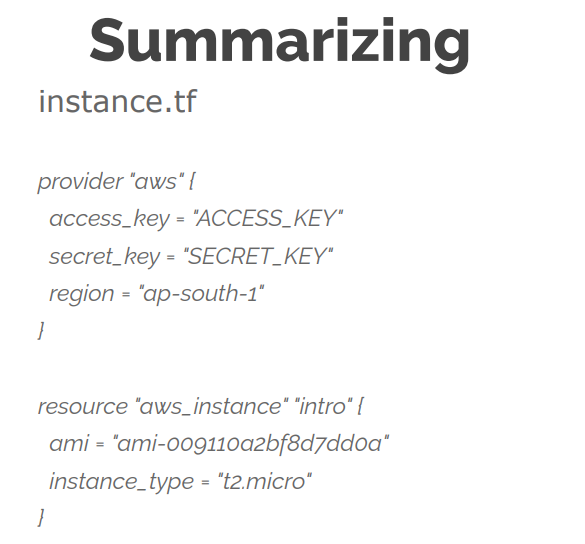
525 cat first\_instances.tf

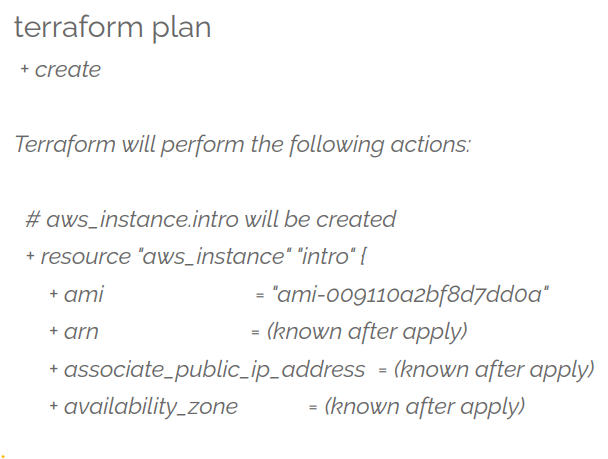
526 terraform apply

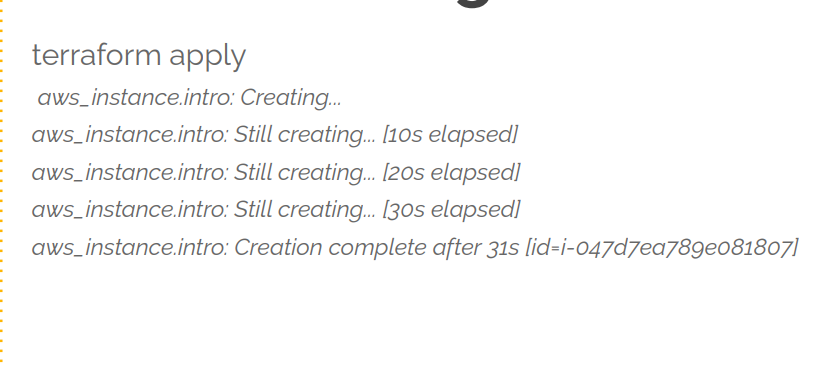
527 ls

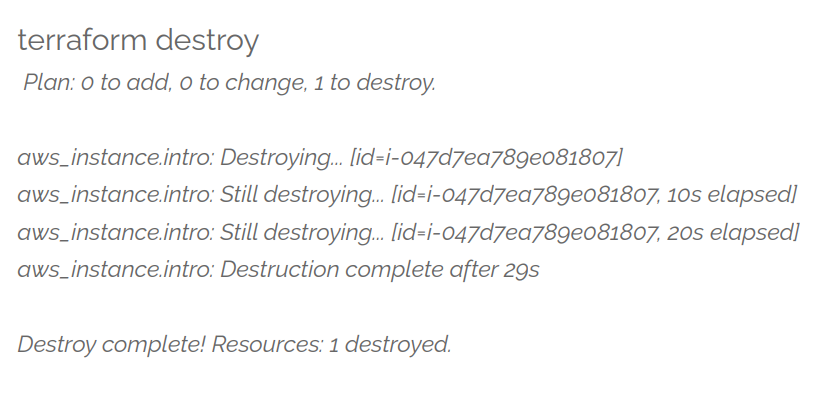
528 terraform destroy

529 history









Absolutely! The provided text is a comprehensive walkthrough of using Terraform for infrastructure provisioning and management. It delves into the basic understanding, setup, and practical application of Terraform, specifically focusing on setting up an EC2 instance on AWS.

### Introduction to Terraform:

Terraform is an "infrastructure as code" tool, allowing users to define and manage infrastructure resources through code. It supports multiple cloud providers like AWS, Google Cloud, Azure, etc.

#### **Key Concepts:**

1. **Provider Configuration**: The configuration for the cloud provider (in this case, AWS) is defined in Terraform, specifying the region, access keys, etc.
2. **Resource Definition**: Resources (such as EC2 instances) are defined in Terraform files using resource blocks that contain attributes like AMI ID, instance type, security groups, etc.

### Practical Walkthrough:

The walkthrough covers these essential steps:

1. **Setting up Terraform**:
   * Installation on different operating systems (Windows, Linux, macOS) via direct download, manual extraction, or using package managers like Choco (for Windows) or Brew (for macOS).
2. **Configuring AWS Access**:
   * Setting up AWS CLI to manage access keys securely.
3. **Writing Terraform Code**:
   * Creating Terraform files (ending with **.tf**) where provider configurations and resource definitions are detailed.
4. **Initialization and Validation**:
   * Running **terraform init** to initialize the project and download required plugins.
   * Using **terraform validate** to check for syntax errors.
   * Applying **terraform fmt** to format the code properly.
5. **Planning and Execution**:
   * Running **terraform plan** to preview the changes before applying them.
   * Finally, executing **terraform apply** to create/modify resources based on the Terraform code.
6. **Monitoring State and Changes**:
   * Terraform maintains a state file (**.tfstate**) containing the current state of resources created by Terraform. It compares this state with the desired state described in the code and applies any necessary changes.
7. **Destroying Resources**:
   * Using **terraform destroy** to delete created resources and revert the infrastructure to its initial state.

### Importance of Each Step:

* **Init and Validation**: Ensures the project is correctly set up and free from syntax errors before making any changes to the infrastructure.
* **Planning and Execution**: Provides a preview of changes to be made, avoiding accidental deletions or unwanted modifications to existing resources.
* **State Maintenance**: Terraform's ability to maintain state ensures consistency between the desired infrastructure state defined in code and the actual state of deployed resources.
* **Destruction of Resources**: Essential for cleaning up created resources and avoiding unnecessary costs or clutter in the cloud environment.

This walkthrough aims to guide users through the entire lifecycle of using Terraform—from setting up, writing code, planning changes, executing those changes, and finally cleaning up the created resources.

# Variable

Certainly! The text you provided covers using variables in Terraform, a tool used for infrastructure as code. Let's break it down into smaller parts:

### Purpose of Variables in Terraform

Variables in Terraform are used to store values that may change across different environments or projects without altering the code directly. They serve multiple purposes:

1. **Sensitive Data Handling:** Variables help in moving critical or confidential data, like access keys or secret keys, out of the scripts to maintain security.
2. **Dynamic Values:** Values that change based on environments (like regions, AMIs, tags, etc.) can be defined as variables. This way, modifying these values doesn't require direct code changes; you simply alter the variable's value.

### Defining Variables in Terraform

Variables are defined in separate **.tf** files, typically named **vars.tf**. The syntax for defining variables is as follows:

terraform Copy code

variable "variable\_name" {

default = value

}

For instance:

variable "region" {

default = "us-east-1"

}

terraformCopy code

### Securing Sensitive Information

It's crucial to keep sensitive information, such as access keys and secret keys, separate from the main scripts. This information can be stored in a **terraform.tfvars** file for security reasons.

### Using Variables in Terraform Files

Once variables are defined, they can be referenced in Terraform files where resources are set up. For instance, to specify a region dynamically:

terraformCopy code

provider "aws" {

region = var.region

}

### Handling Different Values for Different Regions

Map variables can be used to handle different values for different regions. For example:

terraformCopy code

variable "AMI" {

type = map

default = {

"us-west-1" = "ami-12345678"

// Other region-AMI mappings...

}

}

### Applying Variables to Resources

After defining variables, Terraform files for providers (**providers.tf**) and instances (**instance.tf**) should be created to utilize these variables. Always validate changes before applying them using **terraform validate** and **terraform plan**.

### Resource Management

It's important to manage resources properly by using **terraform destroy** to clean up resources created by Terraform. This prevents unnecessary deployments and keeps your environment clean.

By following these steps and best practices, Terraform enables more flexible and secure management of infrastructure by using variables effectively.

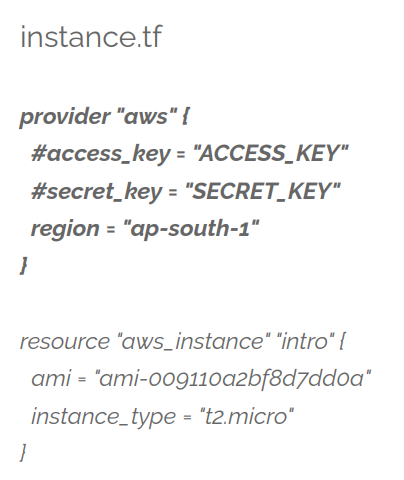
## Variables

➔ Move secrets to another file

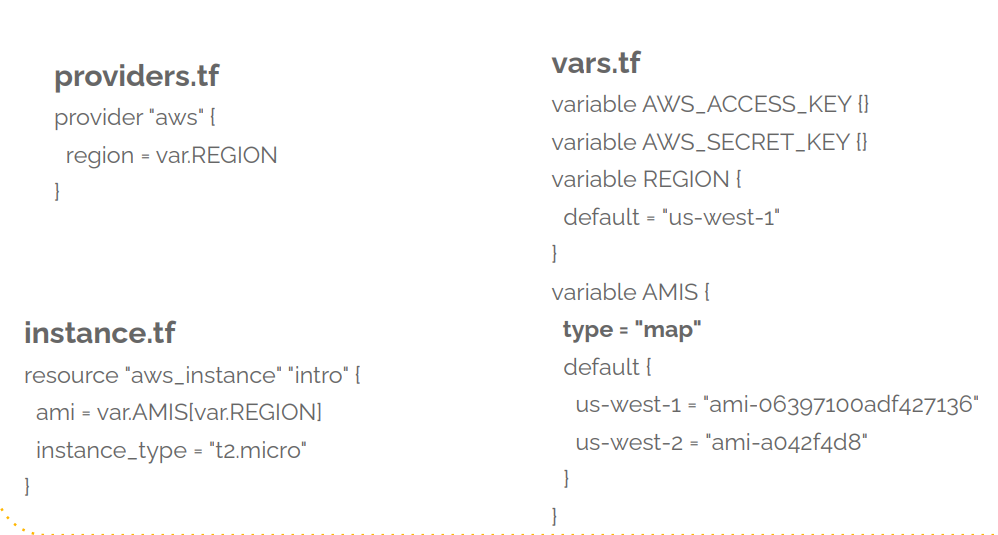
➔ Values that change

◆ AMI, tags, keypair etc

➔ Reuse your code







## Exercise

➔ Write providers.tf file

➔ Write vars.tf file (where we mention variable)

➔ Write instance.tf file (where we access the variable)

➔ Apply Changes (first we’ll validate and then apply the changes)

➔ Make some changes to instance.tf file (to see the changes to see the state)

➔ Apply changes