# 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

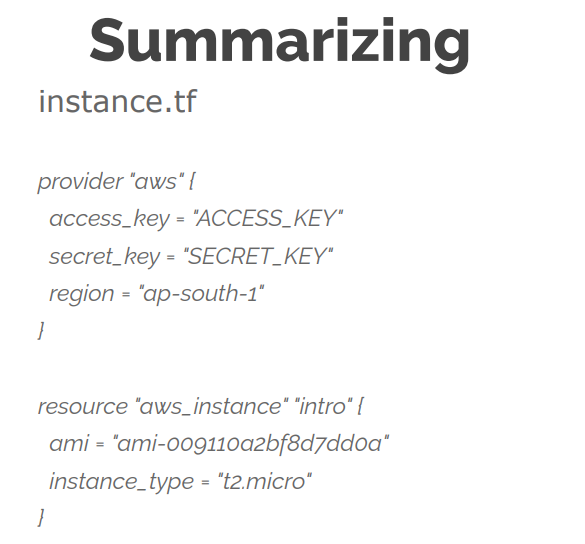
525 cat first\_instances.tf

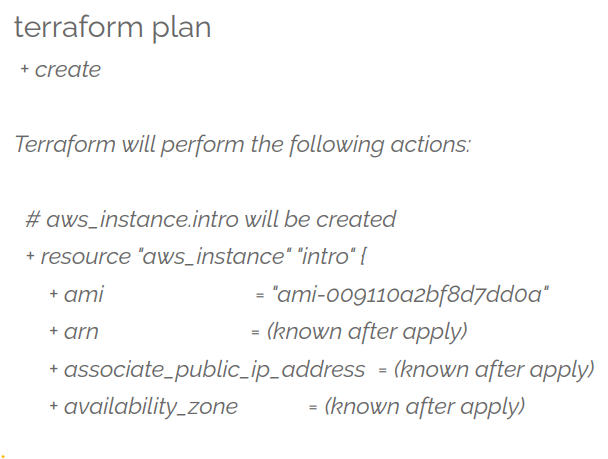
526 terraform apply

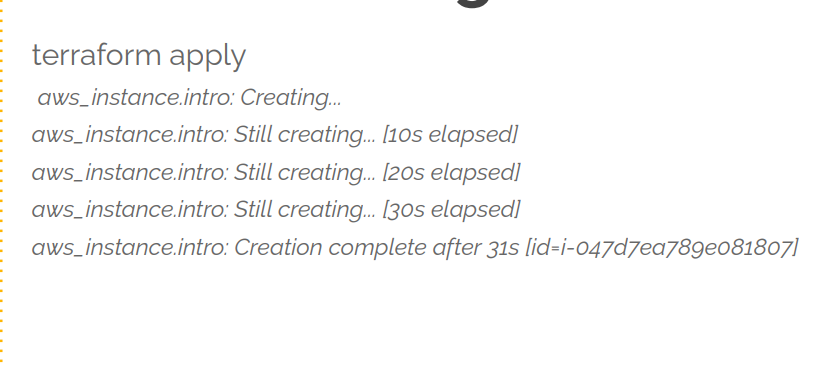
527 ls

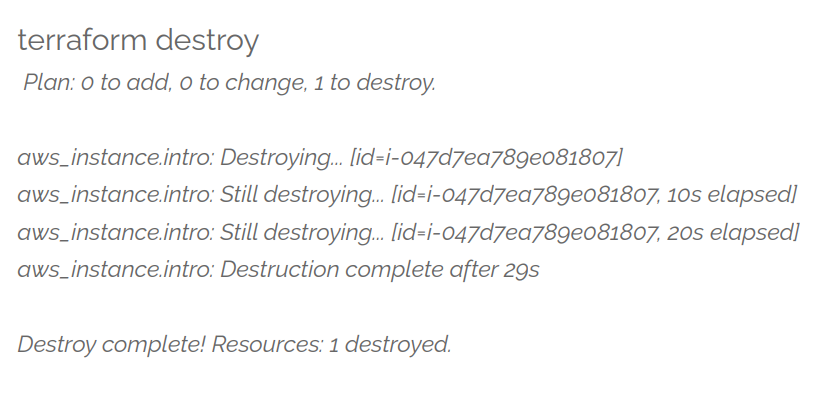
528 terraform destroy

529 history









Welcome to the introductory session on Terraform.

In this session, we'll cover the basics of Terraform and then see how we can use it for project implementation. If you're already familiar with Terraform, you can skip directly to the project video. If you're new to Terraform, continue with this video. Let's get started.

## What is Terraform?

Terraform is an Infrastructure as Code (IaC) tool that allows you to manage your infrastructure through code. You can set up infrastructure on cloud computing environments like AWS, Google Cloud, and Azure, and Terraform ensures that your systems remain in the defined state.

### Comparison with Other Tools

Terraform differs from configuration management tools like Ansible, Puppet, and Chef. These tools focus on managing the state of the operating system, installing packages, and configuring files. While they can automate cloud infrastructure, they do not maintain the state of the infrastructure. Terraform, on the other hand, automates the infrastructure itself and maintains its state, providing a centralized configuration management system for your cloud infrastructure.

### Benefits of Terraform

* **State Management**: Terraform maintains the state of your infrastructure, ensuring consistency.
* **Multi-Cloud Support**: It supports multiple cloud providers like AWS, Google Cloud, Azure, and DigitalOcean.
* **Syntax**: Terraform uses a domain-specific language (DSL) similar to JSON, which is easy to understand.

## Getting Started with Terraform

### Installation

Installing Terraform is straightforward. You can download a binary file for your operating system (Linux, Windows, Mac). Alternatively, you can use package managers like choco for Windows or brew for Mac.

* **Windows**: Use choco install terraform.
* **Mac**: Use brew install terraform.
* **Linux**: Download and extract the binary to /usr/local/bin.

### Setting Up Terraform

1. **Download Terraform**: Extract the binary and add it to your system's PATH.
2. **AWS CLI**: Install AWS CLI to configure your AWS credentials. This is more secure than defining credentials directly in Terraform files.
   * **Windows**: Use choco install awscli.
   * **Mac**: Use brew install awscli.

### AWS Configuration

Create an IAM user in AWS with programmatic access and administrator permissions. Configure AWS CLI with aws configure and enter the access key, secret key, region, and output format.

## Writing Your First Terraform Script

### Directory Structure

Create a directory for your Terraform scripts, e.g., TerraformScripts, and inside it, create a subdirectory for each exercise.

### Example: Launching an EC2 Instance

1. **Create a Terraform file**: instance.tf
2. **Define the provider**:

hcl

Copy code

provider "aws" {

region = "us-east-2"

}

1. **Define the resource**:

hcl

Copy code

resource "aws\_instance" "intro" {

ami = "ami-0abcdef1234567890"

instance\_type = "t2.micro"

tags = {

Name = "DevInstance"

}

}

1. **Initialize Terraform**: terraform init
2. **Validate the configuration**: terraform validate
3. **Format the configuration**: terraform fmt
4. **Plan the execution**: terraform plan
5. **Apply the configuration**: terraform apply
6. **Destroy the infrastructure**: terraform destroy

### Steps in Detail

* **terraform init**: Initializes the Terraform configuration and downloads the AWS provider plugin.
* **terraform validate**: Validates the syntax of the Terraform files.
* **terraform fmt**: Formats the Terraform configuration files.
* **terraform plan**: Shows the execution plan without making any changes.
* **terraform apply**: Applies the changes required to reach the desired state of the configuration.
* **terraform destroy**: Destroys the resources defined in the configuration.

### Example Script

hcl

Copy code

provider "aws" {

region = "us-east-2"

}

resource "aws\_instance" "intro" {

ami = "ami-0abcdef1234567890"

instance\_type = "t2.micro"

tags = {

Name = "DevInstance"

}

}

### Running the Script

1. **Initialize**: terraform init
2. **Validate**: terraform validate
3. **Format**: terraform fmt
4. **Plan**: terraform plan
5. **Apply**: terraform apply
6. **Destroy**: terraform destroy

## Summary

* Terraform is a powerful tool for managing cloud infrastructure as code.
* It maintains the state of your infrastructure and supports multiple cloud providers.
* The installation and setup are straightforward, and the syntax is easy to learn.
* Always validate, format, and plan before applying changes to avoid unintended consequences.

In the next exercise, we'll explore more advanced features of Terraform and continue building our infrastructure.

Let's get started with Terraform and take control of your infrastructure!

# 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.

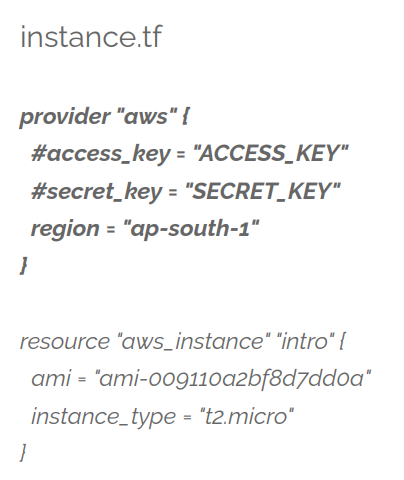
## Variables

➔ Move secrets to another file

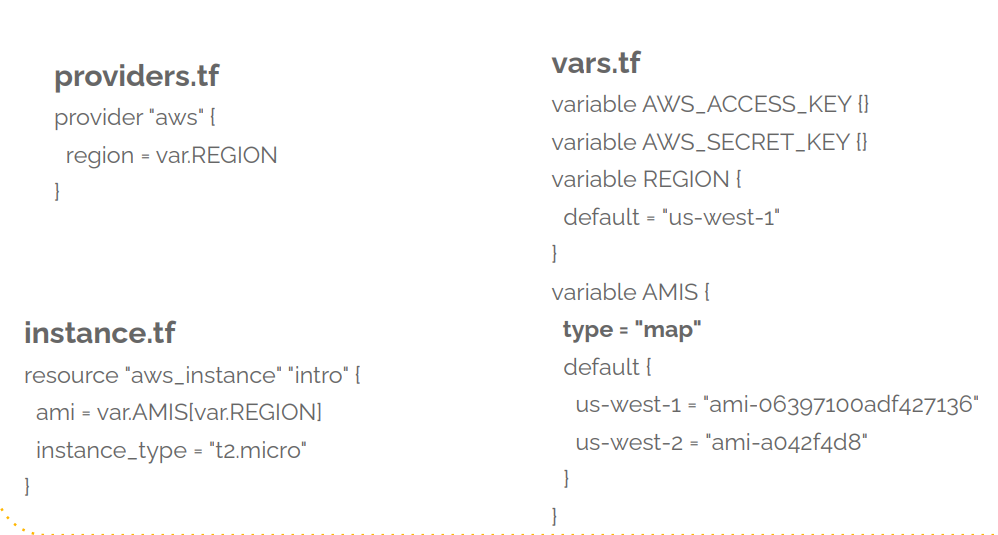
➔ Values that change

◆ 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

It seems like you're working through setting up Terraform configurations for managing AWS resources, specifically focusing on variables and their usage. Here are some observations and suggestions based on your script and steps:

1. **Variables in Terraform:**
   * You've defined variables using the vars.tf file, specifying defaults for region, availability\_zone, and AMI as a map. This is a good practice to parameterize your infrastructure configuration.
   * Remember, Terraform variables can be defined with default values, which can be overridden either through command-line flags, environment variables, or a terraform.tfvars file.
2. **Provider Configuration:**
   * It's a good practice to isolate provider configurations into their own file (providers.tf), which you've done. This keeps your configuration modular and easier to manage.
3. **Resource Configuration:**
   * Your instance.tf file correctly references the variables defined in vars.tf for AMI and region. This ensures that your infrastructure configuration is dynamic and can adapt based on the provided variables.
4. **Workflow:**
   * You've followed a typical Terraform workflow: terraform init to initialize the directory, terraform validate to check for syntax errors, terraform plan to see the execution plan before applying changes, and finally terraform apply to apply those changes.
   * It's crucial to review the Terraform plan output to understand what resources will be added, changed, or destroyed before applying the changes.
5. **Key Pair and Instance State:**
   * You mentioned the challenge of changing a key pair associated with an instance. Terraform, like AWS itself, requires replacing the instance when changing the key pair due to how EC2 security works.
6. **Cleanup:**
   * Finally, you demonstrated good practice by cleaning up resources with terraform destroy after completing your exercises. This prevents unnecessary costs and resources lingering in your AWS account.

Overall, your approach to using Terraform for infrastructure as code (IaC) seems structured and methodical. If you continue to follow these practices, it will help maintain clarity and manageability in your infrastructure deployments. If you have any specific questions or need further guidance on Terraform or AWS, feel free to ask!