Terraform : Infrastructure as Code

Chapter 1: Intro to Terraform

The Rise of DevOps !!

DevOps is a set of practices that combines software development (Dev) and information-technology operations (Ops) which aims to shorten the systems development life cycle and provide continuous delivery with high software quality.

What is Infrastructure

Information technology infrastructure is defined broadly as a set of information technology (IT) components that are the foundation of an IT service; typically physical components (computer and networking hardware and facilities), but also various software and network components.

Infrastructure as a Code !

Infrastructure as Code” (IaC) enables us to describe our infrastructure and applications in source code.

This enables us to treat our infrastructure like we treat our applications and take advantage of development practices and tools

Categories on IAC

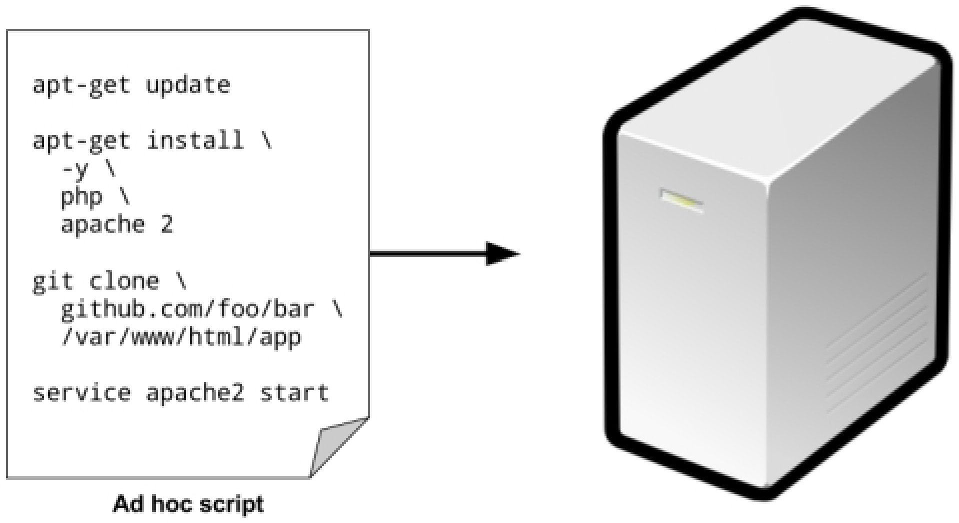
- Ad hoc scripts
- Configuration Management Tools
- Server Templating tools
- Orchestration tools

Adhoc Scripts

The most straightforward approach to automating anything is to write an ad hoc script.

You take whatever task you were doing manually, break it down into discrete steps, use your favorite scripting language (e.g. Bash, Ruby, Python) to define each of those steps in code, and execute that script on your server,

Ad hoc Script

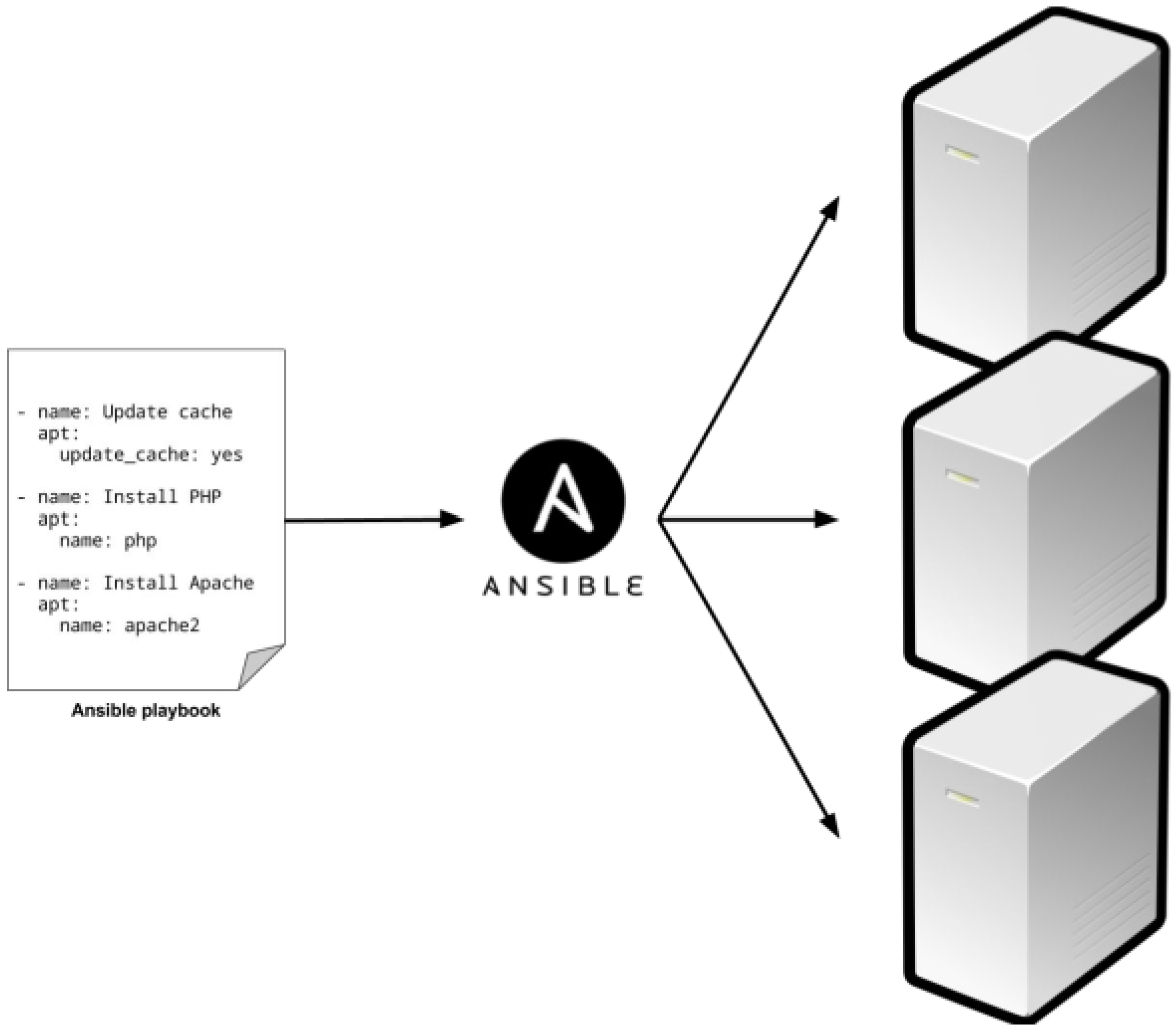


```
apt-get update

apt-get install \
  -y \
  php \
  apache 2

git clone \
  github.com/foo/bar \
  /var/www/html/app

service apache2 start
```

Configuration Management Tools

Chef, Puppet, Ansible, and SaltStack are all configuration management tools, which means they are designed to install and manage software on existing servers

Configuration management tool



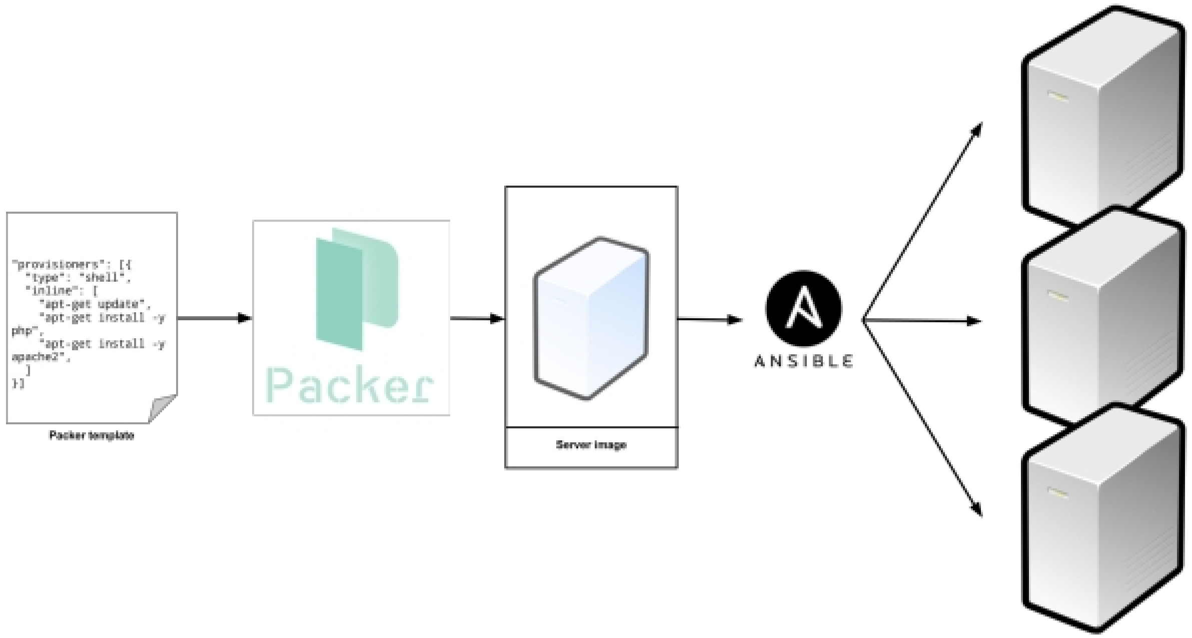
```
- name: Update cache
  apt:
    update_cache: yes

- name: Install PHP
  apt:
    name: php

- name: Install Apache
  apt:
    name: apache2
```

Server templating tools

An alternative to configuration management that has been growing in popularity recently are server templating tools such as Docker, Packer, and Vagrant.

Instead of launching a bunch of servers and configuring them by running the same code on each one, the idea behind server templating tools is to create an image of a server that captures a fully self-contained “snapshot” of the operating system, the software, the files, and all other relevant details.

Server templating tools

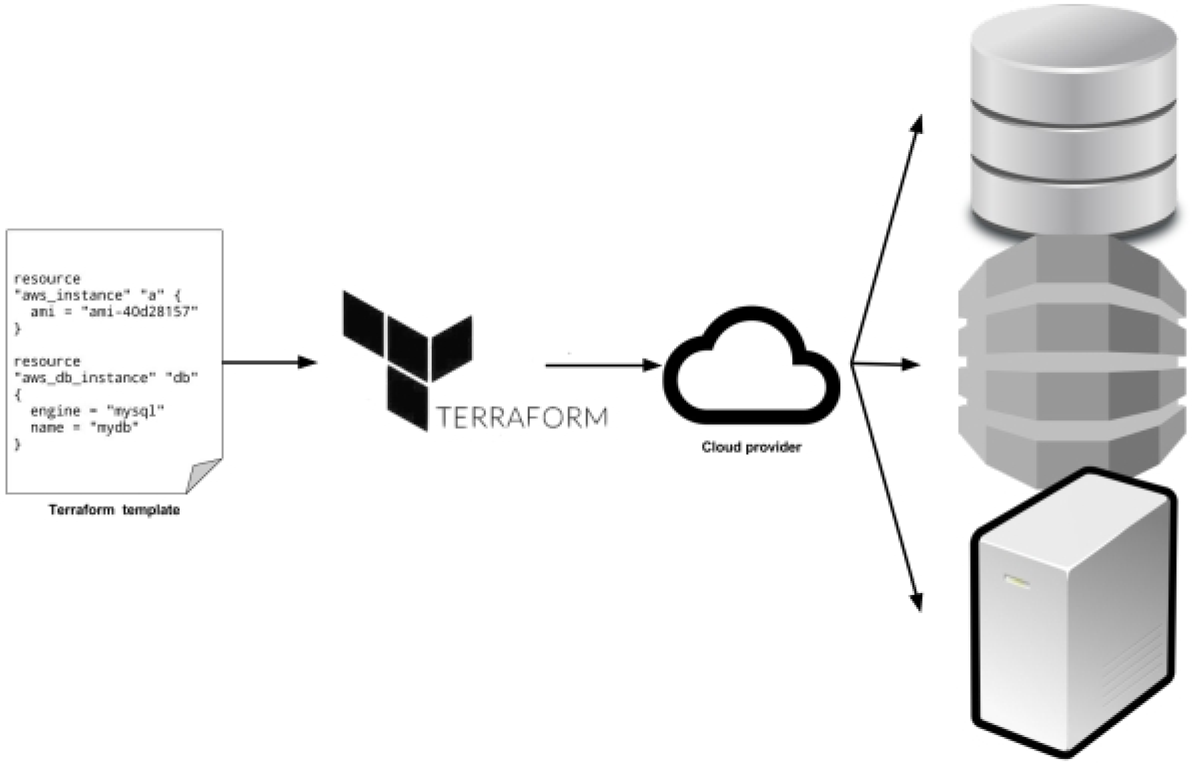


```
"provisioners": [{
  "type": "shell",
  "inline": [
    "apt-get update",
    "apt-get install -y
php",
    "apt-get install -y
apache2",
  ]
}]
```

Orchestration Tools

Whereas configuration management and server templating tools define the code that runs on each server, orchestration tools such as Terraform, CloudFormation, and OpenStack Heat are responsible for creating the servers themselves, a process known as orchestration or provisioning

Orchestration Tools



```
resource
"aws_instance" "a" {
  ami = "ami-40d28157"
}

resource
"aws_db_instance" "db"
{
  engine = "mysql"
  name = "mydb"
}
```

Benefits of IAC

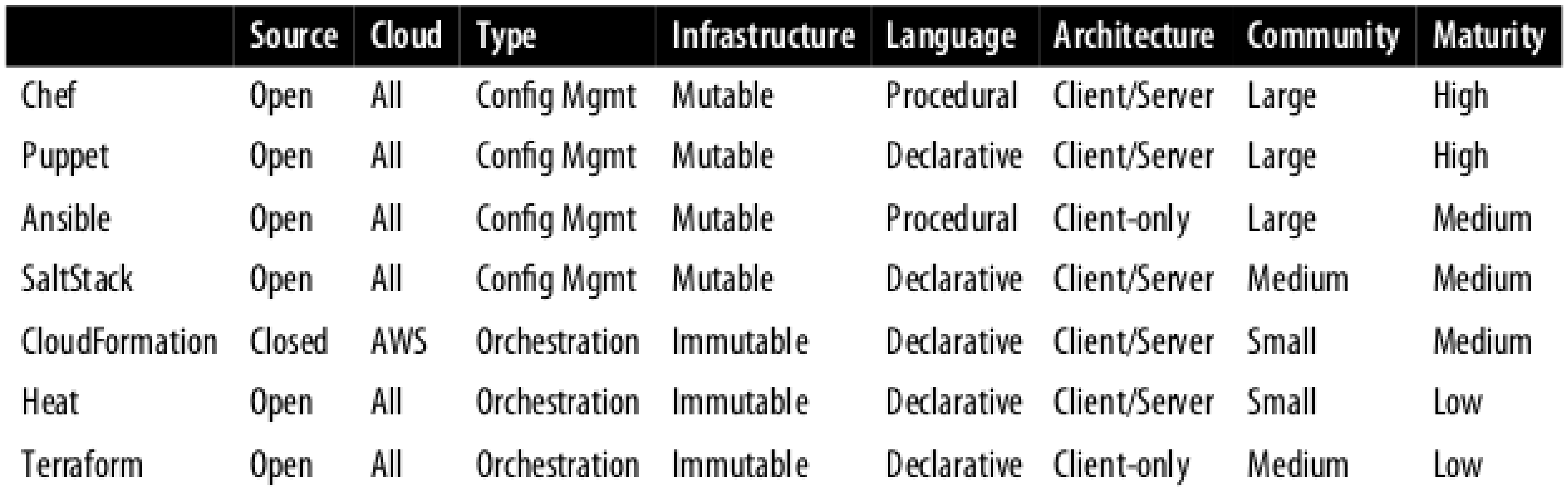
- Self Service
- Speed and Safety
- Documentation
- Version Control
- Validation
- Reuse
- Happiness

Terraform ?

Terraform is an open source tool written in the Go programming language. The Go code compiles down into a single binary (or rather, one binary for each of the supported operating systems) called, not surprisingly, terraform .

You can use this binary to deploy infrastructure from your laptop or a build server or just about any other computer, and you don’t need to run any extra infrastructure to make that happen.

Comparison



	Source	Cloud	Type	Infrastructure	Language	Architecture	Community	Maturity
Chef	Open	All	Config Mgmt	Mutable	Procedural	Client/Server	Large	High
Puppet	Open	All	Config Mgmt	Mutable	Declarative	Client/Server	Large	High
Ansible	Open	All	Config Mgmt	Mutable	Procedural	Client-only	Large	Medium
SaltStack	Open	All	Config Mgmt	Mutable	Declarative	Client/Server	Medium	Medium
CloudFormation	Closed	AWS	Orchestration	Immutable	Declarative	Client/Server	Small	Medium
Heat	Open	All	Orchestration	Immutable	Declarative	Client/Server	Small	Low
Terraform	Open	All	Orchestration	Immutable	Declarative	Client-only	Medium	Low

Chapter 2: Setting up Terraform

Install Terraform ?

Terraform is shipped as a single binary file. The Terraform site contains zip files containing the binaries for specific platforms. Currently Terraform is supported on:

- **Linux: 32-bit, 64-bit, and ARM.**
- **Max OS X: 32-bit and 64-bit.**
- **FreeBSD: 32-bit, 64-bit, and ARM.**
- **OpenBSD: 32-bit and 64-bit**
- **Illumos Distributions: 64-bit**
- **Microsoft Windows: 32-bit and 64-bit**

Install on Linux

To install Terraform on a 64-bit Linux host we can download the zipped binary file. We can use wget or curl to get the file from the download site.

Unpack the terraform binary from the zip file, move it somewhere useful, and change its ownership to the root user.

Test it using “terraform version” command.

Install on Windows

To install Terraform on Microsoft Windows we need to download the terraform executable and put it in a directory.

Create a directory for the executable using Powershell.

Download the terraform executable from the site into the C:\ terraform directory:

Unzip the executable using a tool like 7-Zip into the C:\ terraform directory. Finally, add the C:\terraform directory to the path. This will allow Windows to find the executable.

Terraform Commands

The core of Terraform’s functionality is provided by the terraform binary. Three basic commands to work with is:

- PLAN : Shows us what changes Terraform will make to our infrastructure.
- APPLY : Applies changes to our infrastructure.
- DESTORY : Destroys infrastructure built with Terraform.

Sample Terraform File

```
provider "aws" {

access_key = "your_access_key"

secret_key = "your_secret_key"

region = "ap-south-1"

}

resource "aws_instance" "test-instance" {

ami = "ami-0470e33cd681b2476"

instance_type = "t2.micro"

}
```

Provider

Providers connect Terraform to the infrastructure you want to manage—for example, AWS, Microsoft Azure, or a variety of other Cloud, network, storage, and SaaS services.

They provide configuration like connection details and authentication credentials. You can think about them as a wrapper around the services whose infrastructure we wish to manage.

To see the list of terraform provider, please go to the following link

https://www.terraform.io/docs/providers/index.html

Resource

Resources are the bread and butter of Terraform.

They represent the infrastructure components you want to manage: hosts, networks, firewalls, DNS entries, etc. The resource object is constructed of a type, name, and a block containing the configuration of the resource.

Terraform Init

The terraform init command is used to initialize a working directory containing Terraform configuration files.

This is the first command that should be run after writing a new Terraform configuration or cloning an existing one from version control. It is safe to run this command multiple times.

During init, Terraform searches the configuration for both direct and indirect references to providers and attempts to load the required plugins.

Terraform Plan

The terraform plan command is used to create an execution plan. Terraform performs a refresh, unless explicitly disabled, and then determines what actions are necessary to achieve the desired state specified in the configuration files.

This command is a convenient way to check whether the execution plan for a set of changes matches your expectations without making any changes to real resources or to the state.

Terraform Apply

The terraform apply command is used to apply the changes required to reach the desired state of the configuration, or the pre-determined set of actions generated by a terraform plan execution plan.

By default, apply scans the current directory for the configuration and applies the changes appropriately. However, a path to another configuration or an execution plan can be provided.

Terraform Show

The terraform show command is used to provide human-readable output from a state or plan file.

This can be used to inspect a plan to ensure that the planned operations are expected, or to inspect the current state as Terraform sees it.

Terraform Validate

The terraform validate command validates the configuration files in a directory, referring only to the configuration and not accessing any remote services such as remote state, provider APIs, etc.

Validate runs checks that verify whether a configuration is syntactically valid and internally consistent, regardless of any provided variables or existing state. It is thus primarily useful for general verification of reusable modules, including correctness of attribute names and value types.

Terraform Fmt

The terraform fmt command is used to rewrite Terraform configuration files to a canonical format and style.

Terraform Output

The terraform output command is used to extract the value of an output variable from the state file.

Terraform graph

The terraform graph command is used to generate a visual representation of either a configuration or execution plan.

The output is in the DOT format, which can be used by GraphViz to generate charts.

Terraform Destroy

The terraform destroy command is used to destroy the Terraform-managed infrastructure.

Chapter 3 : Terraform Variables

Terraform Variable

Input variables serve as parameters for a Terraform module, allowing aspects of the module to be customized without altering the module's own source code, and allowing modules to be shared between different configurations.

Types of Variables

- **Strings — String syntax. Can also be Boolean’s: true or false.**
- **Maps — An associative array or hash-style syntax.**
- **Lists — An array syntax.**

Populating Variables

Inputting the variable values every time you plan or apply Terraform configuration is not practical. To address this, Terraform has a variety of methods by which you can populate variables.

1. Loading variables from command line flags.

2. Loading variables from a file.

3. Loading variables from environment variables.

4. Variable defaults.

Chapter 4 : Deploying AWS Insfrastructure

AWS ?

Amazon Web Services (AWS) is a subsidiary of Amazon that provides on-demand cloud computing platforms and APIs to individuals, companies, and governments, on a metered pay-as-you-go basis.

Why we are using AWS ?

AWS is the most popular cloud infrastructure provider, by far.

AWS provides a huge range of reliable and scalable cloud hosting services, including EC2, ASG, ELB etc.

AWS offers a generous Free Tier which should allow you to run all of these examples for free.

Deploying Single Server .. Sample

```
provider "aws" {

access_key = "your_access_key"

secret_key = "your_secret_key"

region = "ap-south-1"

}

resource "aws_instance" "test-instance" {

ami = "ami-0470e33cd681b2476"

instance_type = "t2.micro"
}
```

AWS AMI

The Amazon Machine Image (AMI) to run on the EC2 Instance. You can find free and paid AMIs in the AWS Marketplace or create your own

AWS Instance_type

The type of EC2 Instance to run. Each type of EC2 Instance provides a different amount CPU, memory, disk space, and networking capacity.

The EC2 Instance Types page lists all the available options.

Deploying Process

Use the following commands in sequence to get the server deployed and verify it

- **Terraform init**
- **Terraform plan**
- **Terraform apply**
- **Terraform show**

Deploying the Web Server

Things to understand:

- **User data**
- **Security Group**

Chapter 5 : Terraform States

States?

Every time you run Terraform, it records information about what infrastructure it created in a Terraform state file. By default, when you run Terraform in the folder /foo/bar, Terraform creates the file /foo/bar/terraform.tfstate.

This file contains a custom JSON format that records a mapping from the Terraform resources in your templates to the representation of those resources in the real world.

Working of States

Using the simple JSON format stored locally in the folder, Terraform knows that aws_instance.example corresponds to an EC2 Instance in your AWS account with Id i- 66ba8957.

Every time you run Terraform, it can fetch the latest status of this EC2 Instance from AWS and compare that to what’s in your Terraform templates to determine what changes need to be applied.

Problems with Local State

If you’re using Terraform for a personal project, storing state in a local terra form.tfstate file works just fine.

But if you want to use Terraform as a team on a real product, you run into several problems:

1. Shared Storage for State files.

2. Locking State files

3. Isolating State files.

Remote Stage Storage

With remote state, Terraform writes the state data to a remote data store, which can then be shared between all members of a team.

Terraform supports storing state in Terraform Cloud, HashiCorp Consul, Amazon S3, Alibaba Cloud OSS, and more. Remote state is a feature of backends.

Benefits of remote state

- **Safer storage: Storing state on the remote server helps prevent sensitive information. State file remains same but remote storage like S3 provides a layer to security like making S3 bucket private and giving limited access.**
- **Auditing: Invalid access can be identified by enabling logging.**
- **Share data: Remote storage helps share state file with other members of team.**

Configure S3 for Remote Stage

S3 benefits for remote stage storage are

1. It’s a managed service, so you don’t have to deploy and manage extra infrastruc ture to use it.

2. It’s designed for 99.999999999% durability and 99.99% availability

3. It supports encryption

4. It supports versioning, so every revision of your state file is stored

5. It’s inexpensive, with most Terraform usage easily fitting into the free tier.

Locking Stage file

Terraform provides locking to prevent concurrent runs against the same state. Locking helps make sure that only one team member runs terraform configuration.

Locking helps us prevent conflicts, data loss and state file corruption due to multiple runs on same state file.

Understanding Terragrunt

Terragrunt is a thin wrapper for Terraform that manages remote state for youautomatically and provides locking by using Amazon DynamoDB.

DynamoDB is also part of the AWS free tier, so using it for locking should be free for most teams.

Isolating State file

Isolation via file layout. To get full isolation between environments, you need to: Put the Terraform configuration files for each environment into a separate folder.

Chapter 6 : Terraform Modules

What are modules ?

A module is a container for multiple resources that are used together. Modules can be used to create lightweight abstractions, so that you can describe your infrastructure in terms of its architecture, rather than directly in terms of physical objects.

The .tf files in your working directory when you run terraform plan or terraform apply together form the root module. That module may call other modules and connect them together by passing output values from one to input values of another.

Module Structure

Re-usable modules are defined using all of the same configuration language concepts we use in root modules. Most commonly, modules use:

1) Input variables to accept values from the calling module.

2) Output values to return results to the calling module, which it can then use to populate arguments elsewhere.

3) Resources to define one or more infrastructure objects that the module will manage.

Working of Modules

To define a module, create a new directory for it and place one or more .tf files inside just as you would do for a root module.

Terraform can load modules either from local relative paths or from remote repositories.

Modules can also call other modules using a module block, but we recommend keeping the module tree relatively flat and using module composition as an alternative to a deeply-nested tree of modules, because this makes the individual modules easier to re-use in different combinations.

Modules Input

In Terraform, modules can have input parameters too.

To define them, you use a mechanism you’re already familiar with: input variables.

Declaring Input Variables

Each input variable accepted by a module must be declared using a variable block:

```
variable "image_id" {
  type = string
}


variable "availability_zone_names" {
  type    = list(string)
  default = ["us-west-1a"]
}
```

Modules Output

Output values are like the return values of a Terraform module, and have several uses:

A child module can use outputs to expose a subset of its resource attributes to a parent module.

A root module can use outputs to print certain values in the CLI output after running terraform apply.

When using remote state, root module outputs can be accessed by other configurations via a terraform_remote_state data source.

Declaring Output

Each output value exported by a module must be declared using an output block:

```
output "instance_ip_addr" {
  value = aws_instance.server.private_ip
}
```

The label immediately after the output keyword is the name, which must be a valid identifier. In a root module, this name is displayed to the user; in a child module, it can be used to access the output's value.

File Path Gotchas

By default, Terraform interprets the path relative to the current working directory. That works if you're using the file function in a template that's in the same directory as where you're running terraform apply (that is, if you're using the file function in the root module), but that won't work when you're using file in a module that's defined in a separate folder.

To solve this issue, you can use path.module to convert to a path that is relative to the module folder.

Chapter 7 : Terraform Tips & Tricks

Intro

Terraform is a declarative language. Infrastructure as code in a declarative language tends to provides a more accurate view of what’s actually deployed than a procedural language, so it’s easier to reason about and makes it easier to keep the code base small. However, without access to a full programming language, certain types of tasks become more difficult in a declarative language.

Loop

Terraform offers several different looping constructs, each intended to be used in a slightly different scenario:

- **count parameter: loop over resources.**
- **for_each expressions: loop over resources and inline blocks within a resource.**
- **for expressions: loop over lists and maps.**

Conditionals

Just as Terraform offers several different ways to do loops, there are also several different ways to do conditionals, each intended to be used in a slightly different scenario:

- count parameter: conditional resources.
- for_each and for expressions: conditional resources and inline blocks within a resource.