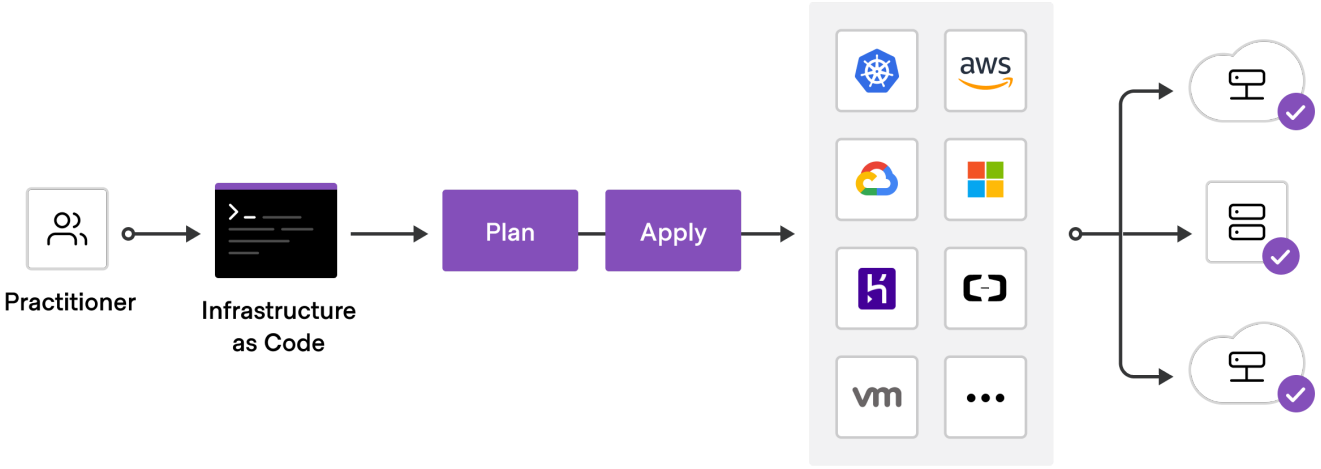
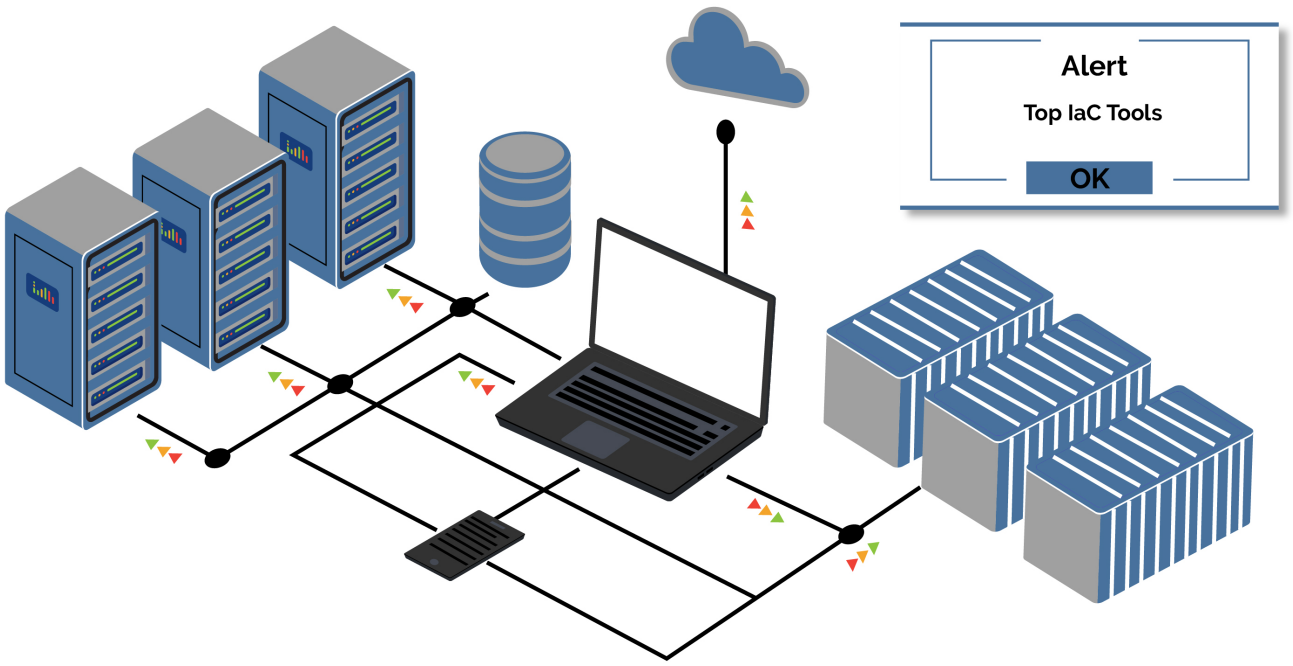
Infrastructure As Code (IaC)

What is infrastructure as code (IaC)?

what is Infrastructure as Code (IaC) ?
It is the management of infrastructure (networks, virtualmachines, load balancers, and connection topology) in a descriptive model, Infrastructure as code (IaC) tools allow you to manage infrastructure with configuration files rather than through a graphical user interface. IaC allows you to build, change, and manage your infrastructure in a safe, consistent, and repeatable way by defining resource configurations that you can version, reuse, and share.





Idempotence is a principle of Infrastructure as Code. Idempotence is the property that a deployment command always sets the target environment into the same configuration, regardless of the environment's starting state.

What is Terraform

Terraform is HashiCorp's infrastructure as code tool.
It lets you define resources and infrastructure in human-readable, declarative configuration files, and manages your infrastructure's lifecycle.
Using Terraform has several advantages over manually managing your infrastructure:

Why Terraform?

- Terraform can manage infrastructure on multiple cloud platforms.
- The human-readable configuration language helps you write infrastructure code quickly.
- Terraform's state allows you to track resource changes throughout your deployments.
- You can commit your configurations to version control to safely collaborate on infrastructure. an IaC model generates the same environment every time it is applied.

> One cool thing about Terraform is, it's plan command lets you see what changes you're about to apply before you apply them.

How Terraform works?

Terraform is logically split into two main parts: Terraform Core and Terraform Plugins.
Terraform Core uses remote procedure calls (RPC) to communicate with Terraform Plugins, and offers multiple ways to discover and load plugins to use.

Terraform core

Terraform Core is a statically-compiled binary written in the Go programming language. The compiled binary is the command line tool (CLI) terraform, the entrypoint for anyone using Terraform.

> The primary responsibilities of Terraform Core are:

1. Infrastructure as code: reading and interpolating configuration files and modules
2. Resource state management
3. Construction of the Resource Graph
4. Plan execution
5. Communication with plugins over RPC

Terraform plugins

Terraform Plugins are written in Go and are executable binaries invoked by Terraform Core over RPC. Each plugin exposes an implementation for a specific service, such as AWS, or provisioner, such as bash. All Providers and Provisioners used in Terraform configurations are plugins. They are executed as a separate process and communicate with the main Terraform binary over an RPC interface.

> The primary responsibilities of Provider Plugins are:

1. Initialization of any included libraries used to make API calls
2. Authentication with the Infrastructure Provider
3. Define Resources that map to specific Services

> The primary responsibilities of Provisioner Plugins are:

- Executing commands or scripts on the designated Resource after creation, or on destruction.

Installation

follow this link and you can easily install terraform on your OS
https://developer.hashicorp.com/terraform/tutorials/aws-get-started/install-cli

---

Terraform providers

Providers
Terraform is used to create, manage, and update infrastructure resources such as physical machines, VMs, network switches, containers, and more. Almost any infrastructure type can be represented as a resource in Terraform.

A provider is responsible for understanding API interactions and exposing resources. Providers generally are an IaaS (e.g. Alibaba Cloud, AWS, GCP, Microsoft Azure, OpenStack), PaaS (e.g. Heroku), or SaaS services (e.g. Terraform Cloud, DNSimple, Cloudflare).

ex : aws provider

provider "aws" {

version = "~> 2.0“

region = "us-east-1"

}

Let's start with AWS

Use the Amazon Web Services (AWS) provider to interact with the many resources supported by AWS. You must configure the provider with the proper credentials before you can use it.

```
terraform {
  required_providers {
    aws = {
      source  = "hashicorp/aws"
      version = "~> 4.0"
    }
  }
}

# Configure the AWS Provider
provider "aws" {
  region = "us-east-1"
}

# Create a VPC
resource "aws_vpc" "example" {
  cidr_block = "10.0.0.0/16"
}
```

Authentication and Configuration

Configuration for the AWS Provider can be derived from several sources, which are applied in the following order:

1. Parameters in the provider configuration
2. Environment variables
3. Shared credentials files & Shared configuration files
4. Container credentials
5. Instance profile credentials and region

1- **Provider Configuration**

```
provider "aws" {
  region     = "us-west-2"
  access_key = "my-access-key"
  secret_key = "my-secret-key"
}
```

2- **Environment variables**

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

```
$ export AWS_ACCESS_KEY_ID="anaccesskey"
$ export AWS_SECRET_ACCESS_KEY="asecretkey"
$ export AWS_REGION="us-west-2"
$ terraform plan
```

3- **Shared credentials files & Shared configuration files** (preferred)
The AWS Provider can source credentials and other settings from the shared configuration and credentials files . By default, these files are located at `$HOME/.aws/config` and `$HOME/.aws/credentials` on Linux and macOS, and `"%USERPROFILE%\.aws\config"` and `"%USERPROFILE%\.aws\credentials"` on Windows.

Contents of the `credentials` file

```
[default]
aws_access_key_id = AKIAIOSFODNN7EXAMPLE
aws_secret_access_key = wJalrXUtnFEMI/K7MDENG/bPxRfiCYEXAMPLEKE

[dev-user]
aws_access_key_id = AKIAI44QH8DHBEXAMPLE
aws_secret_access_key = je7MtGbClwBF/2Zp9Utk/h3yCo8nvbEXAMPLEK
```

The locations of the shared configuration and credentials files can be configured using either the parameters `shared_config_files` and `shared_credentials_files` or the environment variables `AWS_CONFIG_FILE` and `AWS_SHARED_CREDENTIALS_FILE`.

```
provider "aws" {
  shared_config_files      = ["/Users/tf_user/.aws/conf"]
  shared_credentials_files = ["/Users/tf_user/.aws/creds"]
  profile                  = "customprofile"
}
```

Terraform lifecycle

GEEKFLARE

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.

```
terraform init [options]  [DIR]
```

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 plan [options]  [dir]
```

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.

```
terraform apply [options] [dir-or-plan]
```

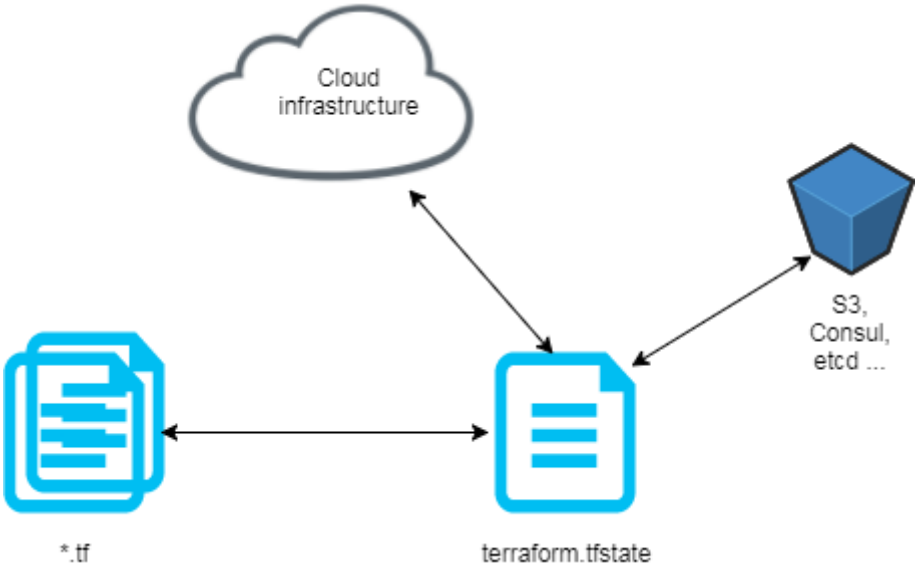
Terraform destroy

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

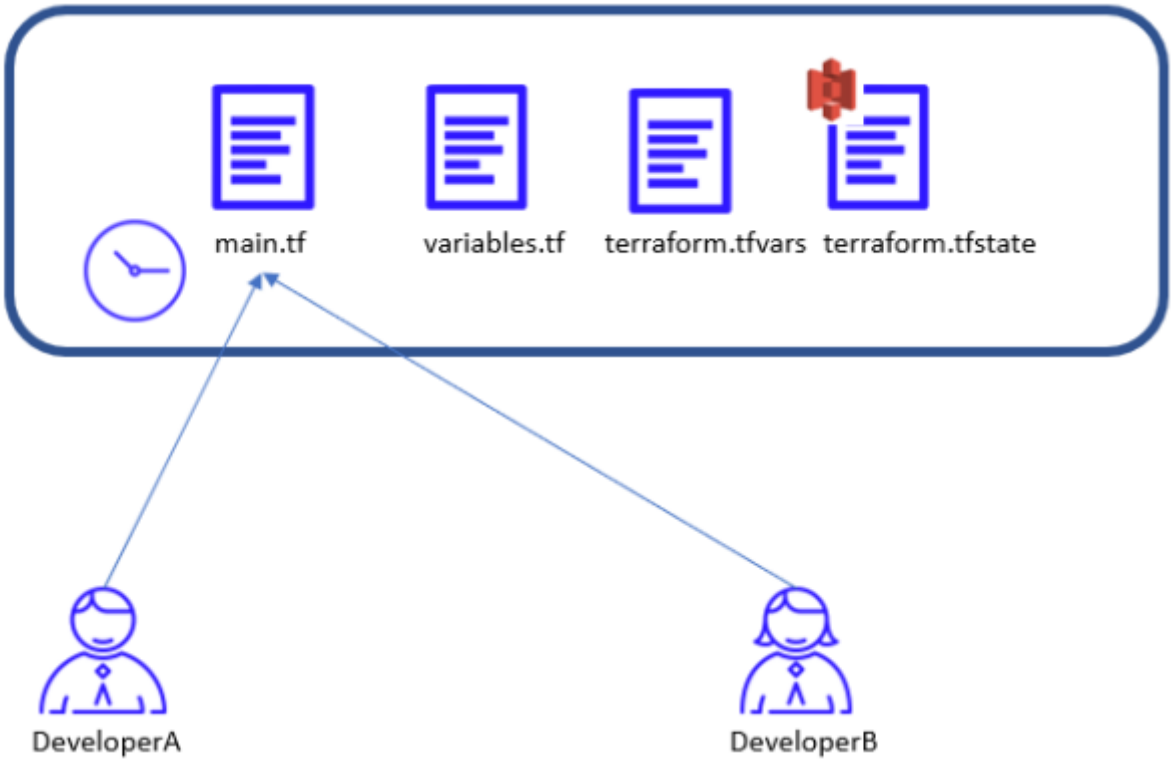
```
terraform destroy [options] [dir]
```

Terraform state

Terraform must store state about your managed infrastructure and configuration. This state is used by Terraform to map real world resources to your configuration, keep track of metadata, and to improve performance for large infrastructures.



This state is stored by default in a local file named "terraform.tfstate", but it can also be stored remotely, which works better in a team environment.
State command



The terraform state command is used for advanced state management. As your Terraform usage becomes more advanced, there are some cases where you may

need to modify the Terraform state. Rather than modify the state directly,
the terraform state commands can be used in many cases instead.

This command is a nested subcommand, meaning that it has further subcommands.
These subcommands are listed to the left.

```
terraform state <subcommand> [options] [args]
```

Locking
If locking is supported by a backend. Then terraform can use remote locking in order to avoid two or more different users running Terraform script at the same time. Terraform will lock your state for all operations that could write state. This prevents others from acquiring the lock and potentially corrupting your state file.

Helpful commands

This command downloads the state from its current location, upgrades the local copy to the latest state file version that is compatible with locally-installed Terraform, and outputs the raw format to stdout.
This is useful for reading values out of state .
It is also useful if you need to make manual modifications to state.

```
terraform state pull
```

The command will list all resources in the state file matching the given addresses (if any). If no addresses are given, all resources are listed.

```
terraform state list [options] [address...]
```

The command will show the attributes of a single resource in the state file that matches the given address.

```
terraform state show [options] ADDRESS
```

Terraform import : terraform is able to import existing infrastructure. This allows you take resources you've created by some other means and bring it under Terraform management.
This is a great way to slowly transition infrastructure to Terraform, or to be able to
be confident that you can use Terraform in the future if it potentially doesn't support
every feature you need today.

```
terraform import resourceName resourceID
```

The command used to remove a binding to an existing remote object without first destroying it, which will effectively make Terraform "forget" the object while it continues to exist in the remote system.

```
terraform state rm resource.Name
```

The command used to retain an existing remote object but track it as a different resource instance address in Terraform, such as if you have renamed a resource block or you have moved it into a different module in your configuration.

```
terraform state  mv [options] SOURCE DESTINATION
```

The terraform fmt : command is used to rewrite Terraform configuration files to a canonical format and style. This command applies a subset of the Terraform language style conventions, along with other minor adjustments for readability.

```
terraform fmt
```

variables

Input variables let you customize aspects of Terraform modules without altering the module's own source code. This functionality allows you to share modules across different Terraform configurations, making your module composable and reusable.

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

> If i wanna use the same code but with different production with it's own variables

```
terraform apply --var-file name.tfvars
```

workspaces

Terraform starts with a single workspace named "default". This workspace is special both because it is the default and also because it cannot ever be deleted. If you've never explicitly used workspaces, then you've only ever worked on the "default" workspace.

Workspaces are managed with the terraform workspace set of commands.

The command is used to list all existing workspaces.

```
terraform workspace list
```

The command is used to choose a different workspace to use for further operations.

```
terraform workspace select
```

The command is used to create a new workspace.

```
terraform workspace new
```

The command is used to delete an existing workspace.

```
terraform workspace delete
```

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.

Resource instances managed by Terraform each export attributes whose values can be used elsewhere in configuration. Output values are a way to expose some of that information to the user of your module.

example :
output "instance_ips" {
value = aws_instance.web.*.public_ip
}

- to list outputs

```
terraform output [options] [NAME]
```

instance_ips = [
"54.43.114.12",
"52.122.13.4",
"52.4.116.53"]

Provisioner

Provisioners can be used to model specific actions on the local machine or on a remote machine in order to prepare servers or other infrastructure objects for service.

```
resource "aws_instance" "web" {
provisioner "local-exec" {
command = "echo The server's IP address is ${self.private_ip}“
}
}
```

modules

A module is a container for multiple resources that are used together.

Every Terraform configuration has at least one module, known as its root module, which consists of the resources defined in the .tf files in the main working directory.

A module can call other modules, which lets you include the child module's resources into the configuration in a concise way. Modules can also be called multiple times, either within the same configuration or in separate configurations, allowing resource configurations to be packaged and re-used.

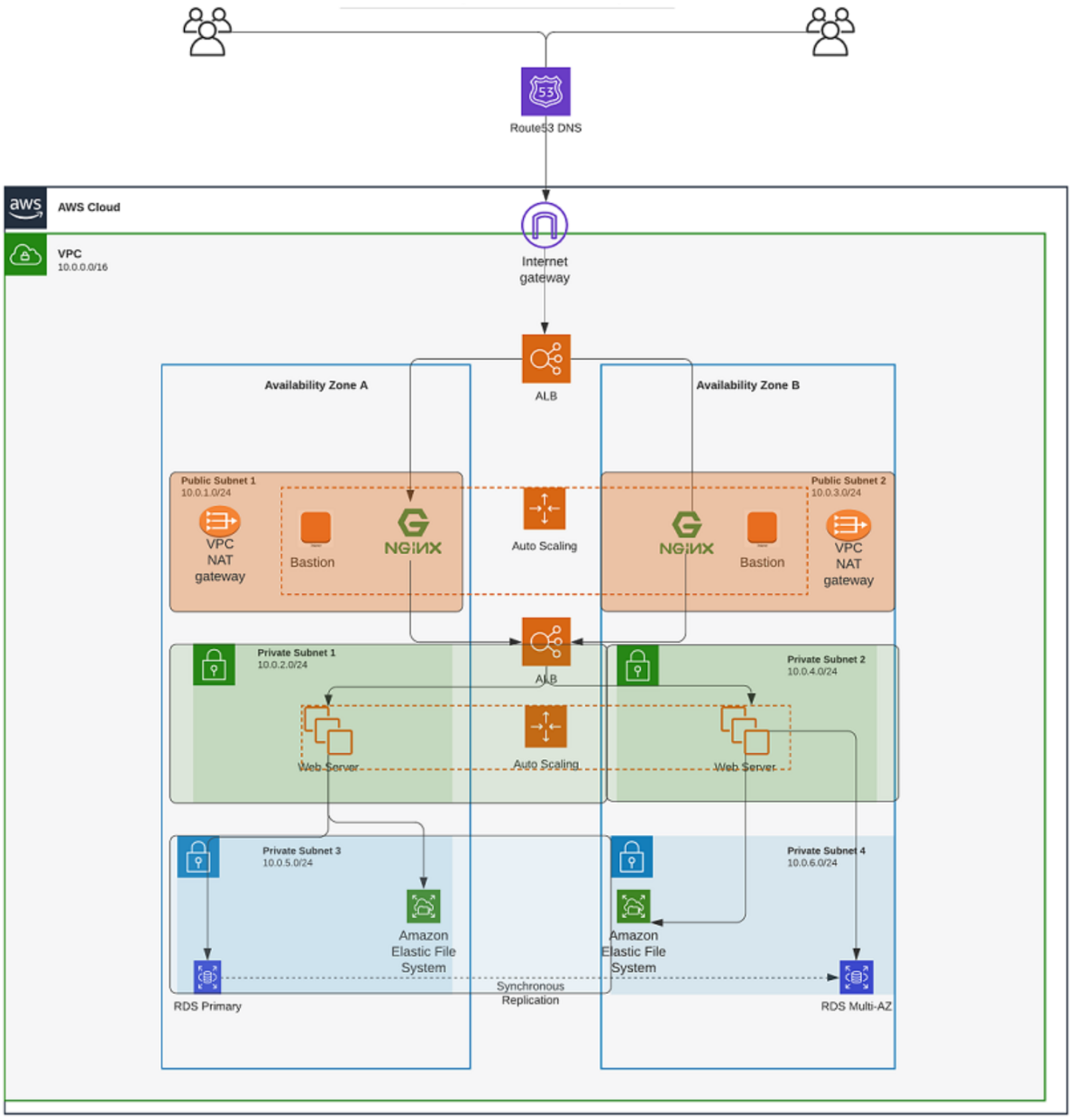
Important meta-argument

`count` is a meta-argument defined by the Terraform language. It can be used with modules and with every resource type.

The `count` meta-argument accepts a whole number, and creates that many instances of the resource or module. Each instance has a distinct infrastructure object associated with it, and each is separately created, updated, or destroyed when the configuration is applied.
taint , untaint

Let's start

mahmoud-sabra/terraform-project (github.com)


Route53 DNS
AWS Cloud
VPC
10.0.0.0/16
Internet gateway
ALB
Availability Zone A
Availability Zone B
Public Subnet 1
10.0.1.0/24
Public Subnet 2
10.0.3.0/24
VPC NAT gateway
Bastion
NGINX
Auto Scaling
NGINX
Bastion
VPC NAT gateway
ALB
Private Subnet 1
10.0.2.0/24
Private Subnet 2
10.0.4.0/24
Web Server
Auto Scaling
Web Server
Private Subnet 3
10.0.5.0/24
Private Subnet 4
10.0.6.0/24
Amazon Elastic File System
Amazon Elastic File System
RDS Primary
Synchronous Replication
RDS Multi-AZ