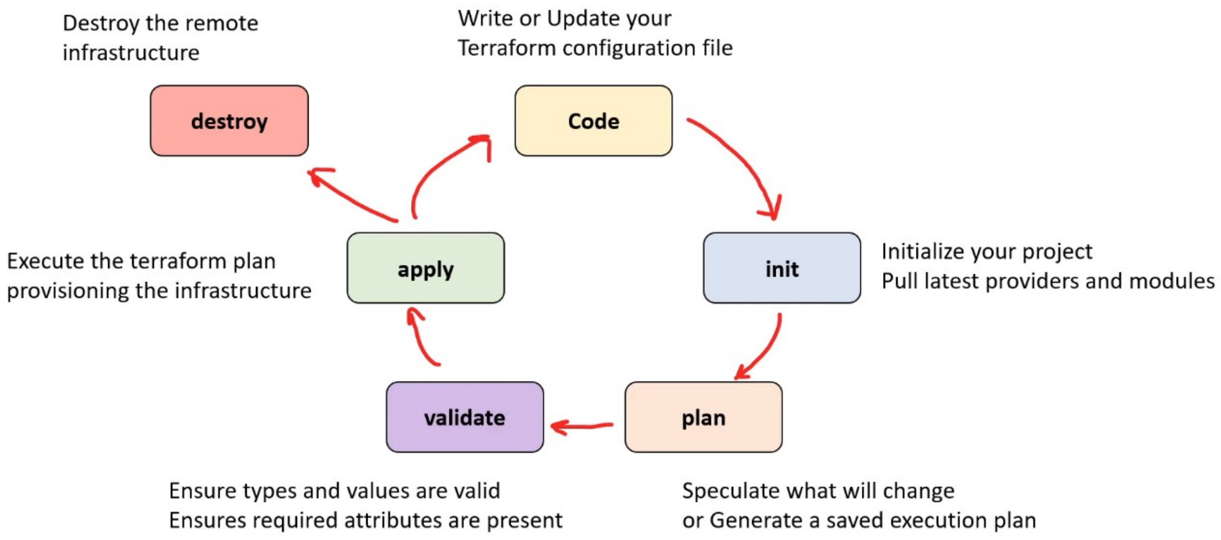
Terraform Lifecycle

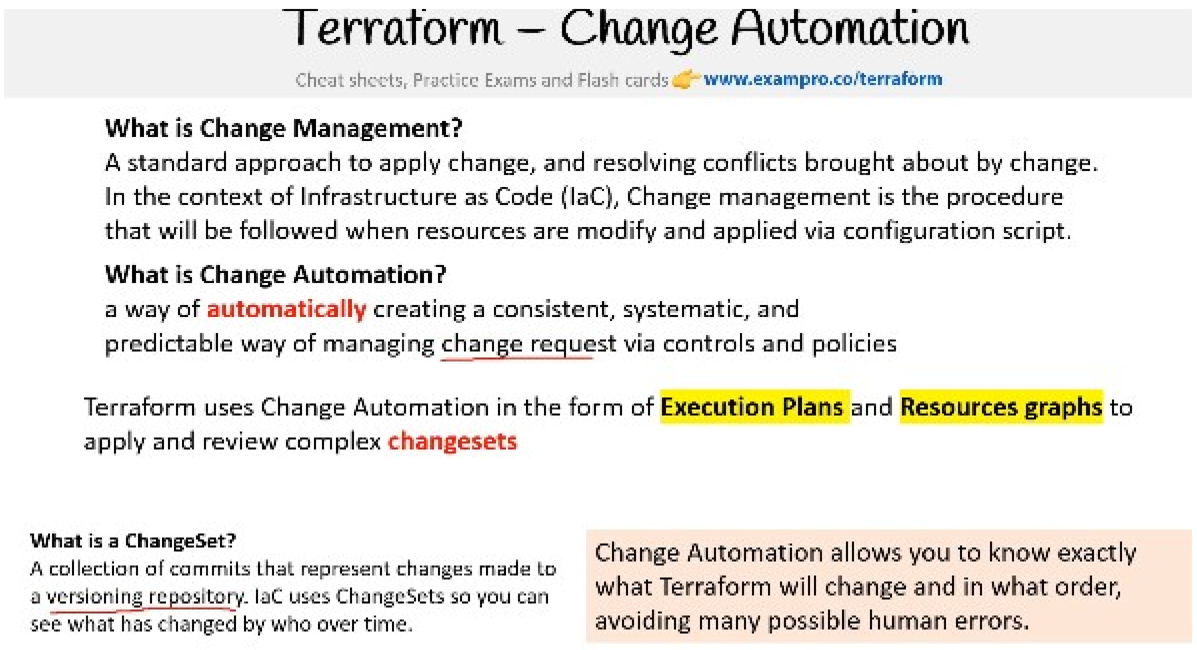
https://www.opensourceforu.com/2020/07/the-benefits-of-using-terraform-as-a-tool-for-infrastructure-as-code-iac/



Destroy the remote infrastructure
Write or Update your Terraform configuration file
destroy
Code
Execute the terraform plan provisioning the infrastructure
apply
init
Initialize your project Pull latest providers and modules
validate
plan
Ensure types and values are valid Ensures required attributes are present
Speculate what will change or Generate a saved execution plan

Change Automation

https://learn.hashicorp.com/tutorials/terraform/automate-terraform



Terraform – Change Automation

Cheat sheets, Practice Exams and Flash cards www.exampro.co/terraform

What is Change Management?
A standard approach to apply change, and resolving conflicts brought about by change. In the context of Infrastructure as Code (IaC), Change management is the procedure that will be followed when resources are modify and applied via configuration script.

What is Change Automation?
a way of **automatically** creating a consistent, systematic, and predictable way of managing change request via controls and policies

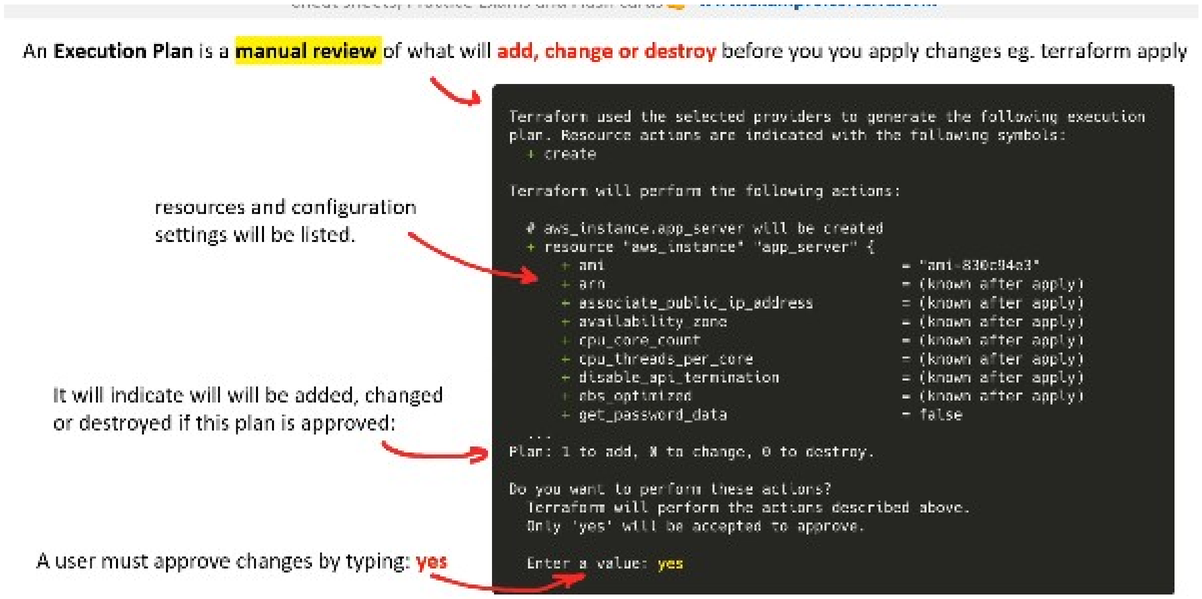
Terraform uses Change Automation in the form of **Execution Plans** and **Resources graphs** to apply and review complex **changesets**

What is a ChangeSet?
A collection of commits that represent changes made to a versioning repository. IaC uses ChangeSets so you can see what has changed by who over time.

Change Automation allows you to know exactly what Terraform will change and in what order, avoiding many possible human errors.

Execution Plans

https://www.terraform.io/intro/index.html - infrastructure-as-code



An **Execution Plan** is a **manual review** of what will **add, change or destroy** before you you apply changes eg. terraform apply

resources and configuration settings will be listed.

It will indicate will will be added, changed or destroyed if this plan is approved:

A user must approve changes by typing: **yes**

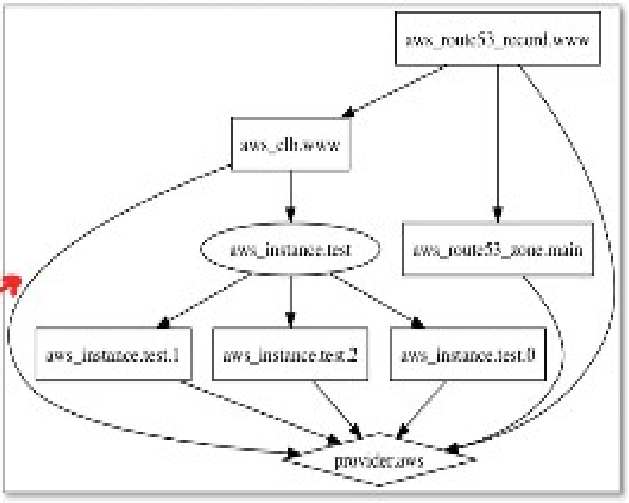
```
Terraform used the selected providers to generate the following execution
plan. Resource actions are indicated with the following symbols:
  + create

Terraform will perform the following actions:

  # aws_instance.app_server will be created
  + resource "aws_instance" "app_server" {
      + ami                          = "ami-830c94e3"
      + arn                          = (known after apply)
      + associate_public_ip_address  = (known after apply)
      + availability_zone            = (known after apply)
      + cpu_core_count               = (known after apply)
      + cpu_threads_per_core         = (known after apply)
      + disable_api_termination      = (known after apply)
      + ebs_optimized                = (known after apply)
      + get_password_data            = false
  ...
Plan: 1 to add, 0 to change, 0 to destroy.

Do you want to perform these actions?
  Terraform will perform the actions described above.
  Only 'yes' will be accepted to approve.

  Enter a value: yes
```

Visualizing Execution Plans

You can **visualize an execution plan** as a graph using the **terraform graph** command
Terraform will output a GraphViz file (you'll need GraphViz installed to view the file)

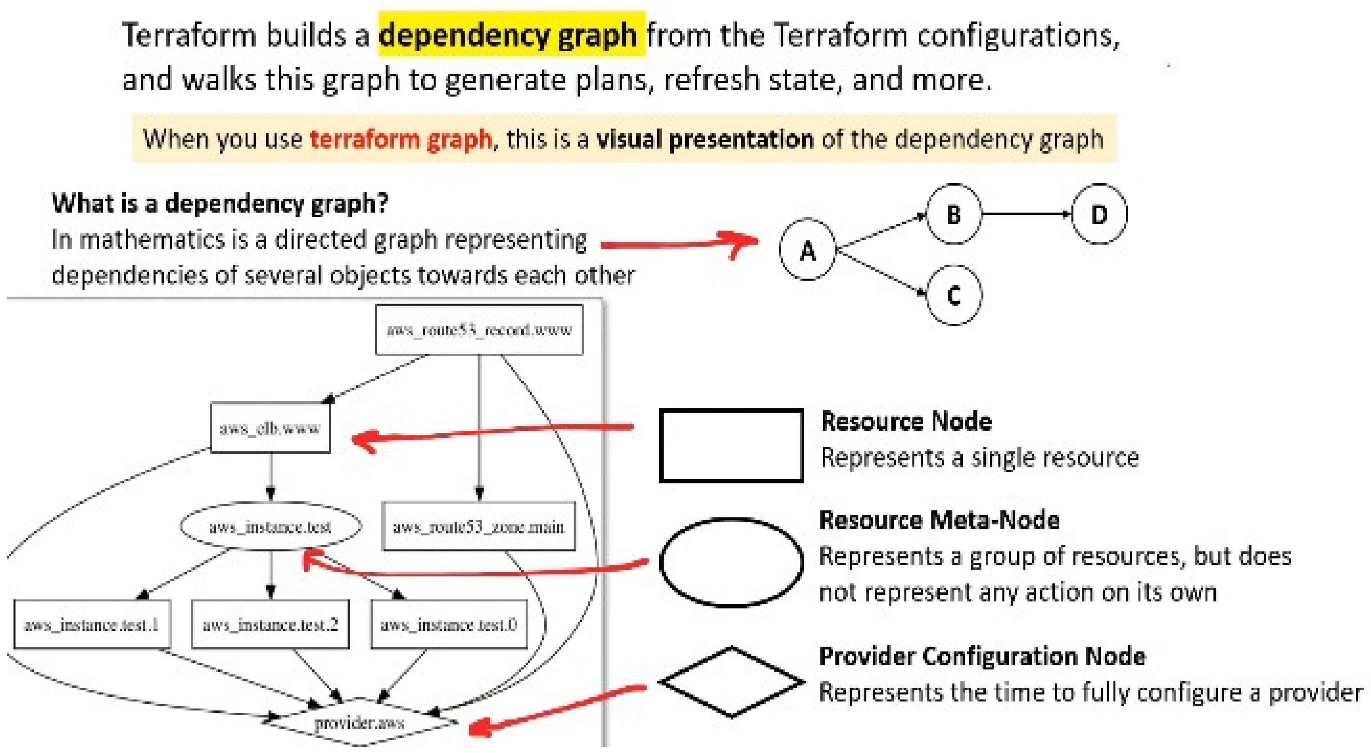
What is GraphViz?
open-source tools for drawing graphs specified in DOT language scripts having the file name extension "gv"

```
terraform graph | dot -Tsvg > graph.svg
```



Resource Graphs

https://www.terraform.io/docs/internals/graph.html


Terraform builds a **dependency graph** from the Terraform configurations, and walks this graph to generate plans, refresh state, and more.

When you use **terraform graph**, this is a **visual presentation** of the dependency graph

What is a dependency graph?
In mathematics is a directed graph representing dependencies of several objects towards each other

Resource Node
Represents a single resource

Resource Meta-Node
Represents a group of resources, but does not represent any action on its own

Provider Configuration Node
Represents the time to fully configure a provider

TF – Use Cases

https://www.terraform.io/intro/use-cases.html - multi-cloud-deployment

IaC for Exotic Providers

Terraform supports a variety of providers outside of GCP, AWS, and Azure and sometimes is the only provider.

Terraform is open-source and extendable to any API that could be used to create IaC tooling for any kind of cloud platform or technology. E.g. Heroku, and Spotify Playlists.

Multi-Tier Applications – Terraform by default makes it easy to divide large and complex applications into isolated configuration scripts (module). It has a complexity advantage over cloud-native IaC tools for its flexibility while retaining simplicity over Imperative tools.

Disposable Environments – Easily stand up an environment for a software demo or a temporary development environment

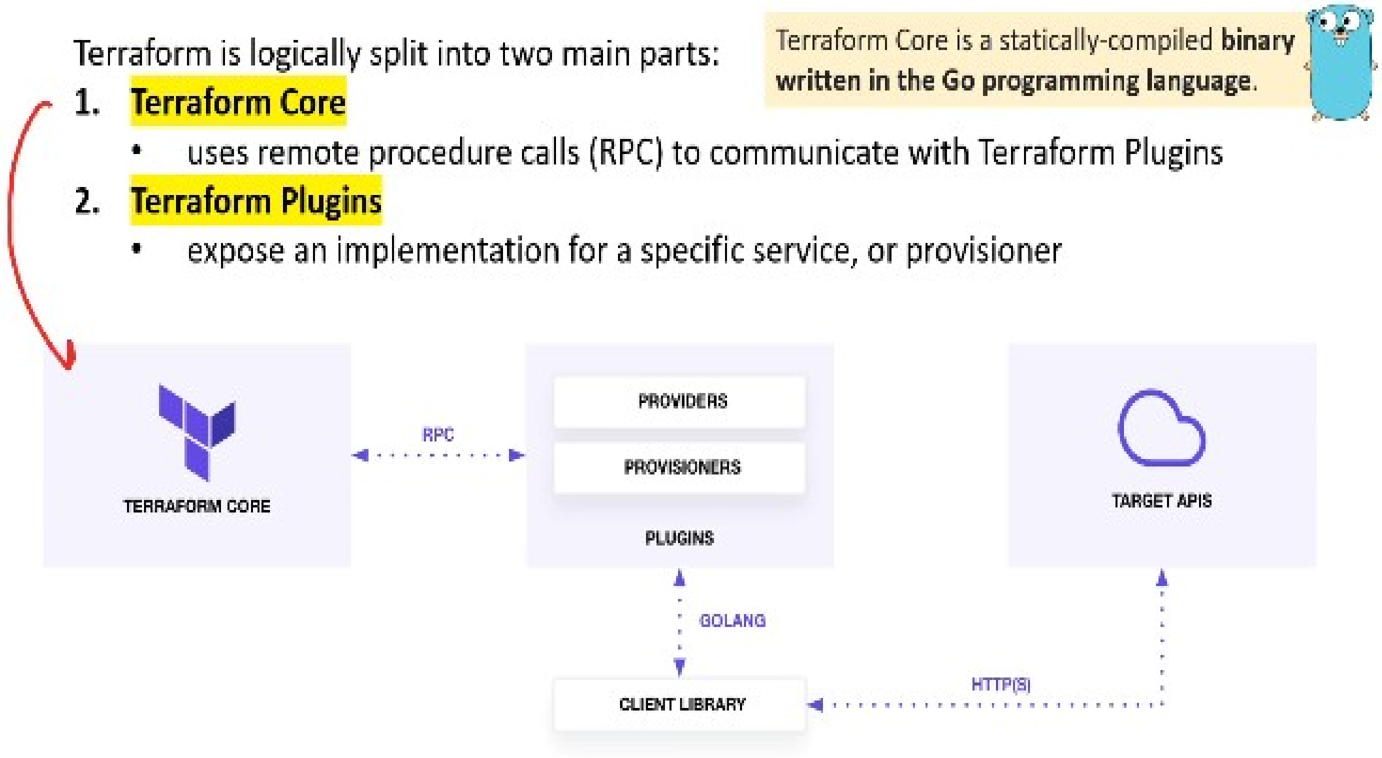
Resource Schedulers – Terraform is not just defined to the infrastructure of cloud resources but can be used to dynamic schedule Docker containers, Hadoop, Spark, and other software tools. You can provision your scheduling grid.

Multi-Cloud Deployment – Terraform is cloud-agnostic and allows a single configuration to manage multiple providers and even handle cross-cloud dependencies.

Terraform Core & Terraform Plugins

https://www.terraform.io/docs/extend/how-terraform-works.html

https://learn.hashicorp.com/tutorials/terraform/provider-use



Terraform is logically split into two main parts:

1. Terraform Core
   - uses remote procedure calls (RPC) to communicate with Terraform Plugins
2. Terraform Plugins
   - expose an implementation for a specific service, or provisioner

Terraform Core is a statically-compiled **binary written in the Go programming language**.

Terraform – UP & RUNNING

https://www.google.ca/books/edition/Terraform_Up_Running/7bytDwAAQBAJ?hl=en&gbpv=1&printsec=frontcover

A Must Read Book

If you want to go beyond this course for things like:

- Testing your Terraform Code

- Zero-Downtime Deployment
- Common Terraform Gotchas
- Composition of Production Grade Terraform Code

Terraform BEST Practices

https://www.terraform-best-practices.com/

Online Open Book : https://www.terraform-best-practices.com/

GITHUB REPO for All Resources : https://github.com/ExamProCo/Terraform-Associate-Labs

Install Terraform : https://learn.hashicorp.com/tutorials/terraform/install-cli

Terraform Commands

A.$terraform init [options]:

Initialize backend and provider plugins.
General Options
The following options apply to all of (or several of) the initialization steps:

- -input=true Ask for input if necessary. If false, will error if input was required.
- **-lock=false Disable locking of state files during state-related operations.**
- -lock-timeout=<duration> Override the time Terraform will wait to acquire a state lock. The default is 0s (zero seconds), which causes immediate failure if the lock is already held by another process.
- -no-color Disable color codes in the command output.
- -upgrade Opt to **upgrade modules and plugins** as part of their respective installation steps and upgrade already installed modules.
- `-backend=false Skip Backend configuration.`
- -force-copy option suppresses these prompts and answers "yes"
- -migrate-state option will attempt **to copy existing state to the new backend**, and depending on what changed

Optionally, init can be run against an empty directory with the -from-module=MODULE-SOURCE option, in which case the given module will be copied into the target directory before any other initialization steps are run.

This special mode of operation supports two use-cases:

- Given a version control source, it can serve as a shorthand for checking out a configuration from version control and then initializing the working directory for it.
- If the source refers to an *example* configuration, it can be copied into a local directory to be used as a basis for a new configuration.

Files created in init :

$find .terraform/ //.terraform is folder

1. .terraform/
2. .terraform/plugins
3. .terraform/plugins/darwin_amd64
4. .terraform/plugins/darwin_amd64/lock.json
5. .terraform/plugins/darwin_amd64/terraform-provider-cloudflare_v1.0.0_x4

B. $terraform fmt :

```
terraform fmt [options] [target...]
```

By default, **fmt scans the current directory for configuration files**. If you provide a directory for the target argument, then fmt will scan that directory instead. **If you provide a file, then fmt will process just that file**. If you provide a single dash (-), then fmt will read from standard input (STDIN).

The command-line flags are all optional. The following flags are available:

- -list=false - Don't list the files containing formatting inconsistencies.
- -write=false - Don't overwrite the input files. (This is implied by -check or when the input is STDIN.)
- -diff - Display diffs of formatting changes
- -check - Check if the input is formatted. Exit status will be 0 if all input is properly formatted. If not, exit status will be non-zero and the command will output a list of filenames whose files are not properly formatted.
- -recursive - Also process files in subdirectories. By default, only the given directory (or current directory) is processed.

C. $terraform validate :

terraform validate [options]

This command accepts the following options:

- -json - Produce output in a machine-readable JSON format, suitable for use in text editor integrations and other automated systems. Always disables color.
- -no-color - If specified, output won't contain any color.

D.$terraform plan :

1. The terraform plan command lets you to preview the actions Terraform would take to modify your infrastructure, or save a speculative plan which you can apply later. The function of terraform plan is speculative: you cannot apply it unless you save its contents and pass them to a terraform apply command. In an automated Terraform pipeline, applying a saved plan file ensures the changes are the ones expected and scoped by the execution plan, even if your pipeline runs across multiple machines.
2. Generate a saved plan with the -out flag. You will review and apply this plan later in this tutorial.

E. $terraform apply :

$terraform apply + **update :** To just modify a resource, go to your .tf config file(s) and update the attributes. Now run $terraform apply and your resource will be modified, not deleted and created again with other configuration. This is called **IDEMPOTENCY**.

Input Variables

Input variables are your primary way to apply parameters to your Terraform configuration. It allows aspects of your configuration to be customized without altering the Terraform module's own source code and allowing modules to be shared between different configurations.

You can declare variables in the root module of your configuration. You can set those by using CLI options and environment variables. If you are familiar with most programming languages, Terraform modules are comparable to function definitions:

- Input variables are like function arguments.
- Output values are like function return values.
- Local values are like a function's temporary local variables.
- An input variable needs to be declared using a `variable block` in the Terraform configuration. The following example is how a `variable block` looks:

- `variable "image" {`

  `default` – Default value which then makes the variable optional.

`type` – Specifies what value types are accepted for the variable. (string, number, bool) (`list(<TYPE>)`, `set(<TYPE>)`,`map(<TYPE>)`,`object({<ATTR_NAME> = <TYPE>, … })`,`tuple([<TYPE>, …])`)

`description` – Specifies the input variable's documentation.

`validation` – Defines validation rules, usually in addition to type constraints.

`sensitive` – This limits the Terraform UI output when the variable is used in the configuration.

```
}
```

```
Referencing Variables  :  attribute = var.variableName
                          Eg : ami = var.image_id
```

There is an order to things! **

There is a variable definition precedence you need to be aware of when declaring variables. Terraform loads variables in a specific order. The order is as follows:

- Environment variables
- The `terraform.tfvars` file (if present)
- The `terraform.tfvars.json` file (if present)
- Any `*.auto.tfvars` or `*.auto.tfvars.json` files (if present)
- Any `-var` and `-var-file` options on the command line.

The many ways to assign variables in Terraform

We have gone over some of the basics about input variables. Now let's take a look at a few examples of how we can declare our variables.

This example below is how you can declare variables from the command line.

```
$ terraform apply -var="image_id=ami-abc123"

$ terraform apply -var='image_id_list=["ami-abc123","ami-
def456"]' -var="instance_type=t2.micro"

$ terraform apply -var='image_id_map={"us-east-1":"ami-
abc123","us-east-2":"ami-def456"}'
```

Here you can see we are providing the variables to `terraform apply` command by the `-var` option to specify our variables. If we have more than a few variables, we can create a variables file with either the `.tfvars` or `.tfvars.json` file extension and then specifying that file on the command line like the below example.

```
$ terraform apply -var-file="testing.tfvars"
```

The variable definition files use the same syntax as the rest of the Terraform configuration files.

We can also use Environment variables to set variables in Terraform. You can simply do this by exporting your variable like the below example.

```
$ export TF_VAR_image_id=ami-abc123
```

Local Values

Hands-on: Try the Simplify Terraform Configuration with Locals tutorial.
A local value assigns a name to an expression, so you can use the name multiple times within a module instead of repeating the expression.

If you're familiar with traditional programming languages, it can be useful to compare Terraform modules to function definitions:

- Input variables are like function arguments.
- Output values are like function return values.
- Local values are like a function's temporary local variables.

Note: For brevity, local values are often referred to as just "locals" when the meaning is clear from context.

Declaring a Local Value

A set of related local values can be declared together in a single locals block:

```
locals {
service_name = "forum"
owner        = "Community Team"
}
```

Copy

The expressions in local values are not limited to literal constants; they can also reference other values in the module in order to transform or combine them, including variables, resource attributes, or other local values:

```
locals {
  # Ids for multiple sets of EC2 instances, merged together
  instance_ids = concat(aws_instance.blue.*.id, aws_instance.green.*.id)
}

locals {
  # Common tags to be assigned to all resources
  common_tags = {
    Service = local.service_name
    Owner   = local.owner
  }
}
```

Using Local Values

Once a local value is declared, you can reference it in expressions as local.<NAME>.

Note: Local values are *created* by a locals block (plural), but you *reference* them as attributes on an object named local (singular). Make sure to leave off the "s" when referencing a local value!

```
resource "aws_instance" "example" {
  # ...

  tags = local.common_tags
}
```

A local value can only be accessed in expressions within the module where it was declared.

When To Use Local Values

Local values can be helpful to avoid repeating the same values or expressions multiple times in a configuration, but if overused **they can also make a configuration hard to read** by future maintainers by hiding the actual values used.

Use local values only in moderation, in situations where a single value or result is used in many places *and* that value is likely to be changed in future**. The ability to easily change the value in a central place is the key advantage of local values.**

Terraform Modules

Find modules in terraform registry.

- Terraform modules are a simpler way to add customized snippets of code to out .tf config files.
- Run $terraform init to install respective modules as in "**source**" attribute in module block. Then run $terraform apply

To change provider in a resource :

resource "resName" "reference" { … provider = "providername" …}

To change a provider in Module :

module "modulename" {

    …

    Providers = { provider = **provider.reference** }

    …

}

Workspaces

Each Terraform configuration has an associated backend that defines how Terraform executes operations and where Terraform stores persistent data, like state.

The persistent data stored in the backend belongs to a workspace. The backend initially has only one workspace containing one Terraform state associated with that configuration. Some backends **support multiple named workspaces**, allowing multiple states to be associated with a single configuration. The configuration still has only one backend, but you can deploy multiple distinct instances of that configuration without configuring a new backend or changing authentication credentials.

Note: The Terraform CLI workspaces are different from workspaces in Terraform Cloud. Refer to Initializing and Migrating for details about migrating a configuration with multiple workspaces to Terraform Cloud.

Terraform starts with **a single, default workspace named `default` that you cannot delete**. If you have not created a new workspace, you are using the default workspace in your Terraform working directory.

When you run `terraform plan` in a new workspace, Terraform does not access existing resources in other workspaces. These resources still physically exist, but you must switch workspaces to manage them.

Workspaces in the Terraform CLI refer to separate instances of state data inside the same Terraform working directory. They are distinctly different from workspaces in Terraform Cloud, which each have their own Terraform configuration and function as separate working directories.

Terraform **relies on state to associate resources with real-world objects**. When you run the same configuration multiple times with separate state data, Terraform can manage multiple sets of non-overlapping resources.

Terraform Cloud vs. Terraform CLI Workspaces

Both Terraform Cloud and Terraform CLI have features called workspaces, but they function differently.

- **Terraform Cloud workspaces are required. They represent all of the collections of infrastructure in an organization.** They are also a major component of role-based access in Terraform Cloud. You **can grant individual users and user groups permissions for one or more workspaces** that dictate whether they can manage variables, perform runs, etc. You cannot manage resources in Terraform Cloud without creating at least one workspace.

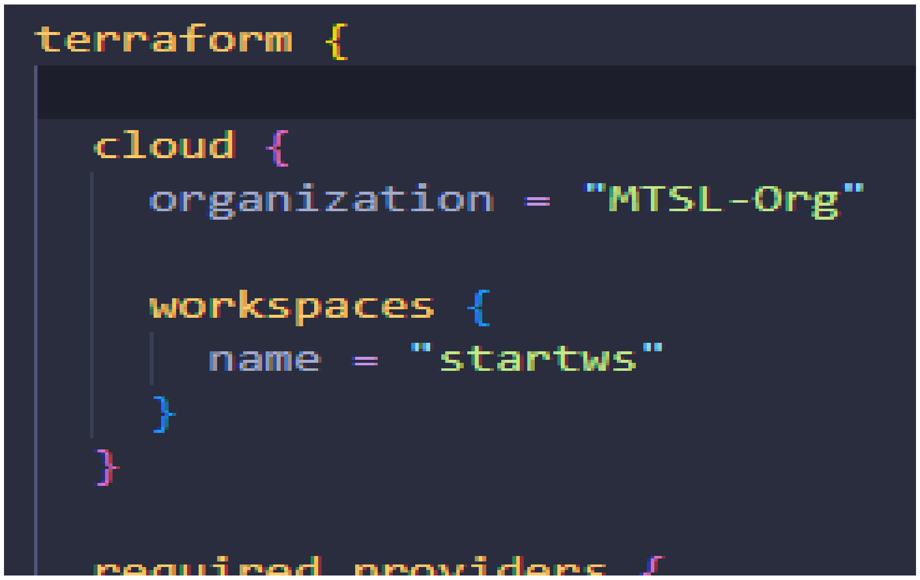
- Terraform CLI workspaces **are associated with a specific working directory and isolate multiple state files in the same working directory,** letting you manage multiple groups of resources with a single configuration. The Terraform CLI does **not require you to create CLI workspaces**. Refer to Workspaces in the Terraform Language documentation for more details.

Managing CLI Workspaces

- Every initialized working directory starts with one workspace named **default**.
- Use the terraform workspace list, terraform workspace new, and terraform workspace delete commands to manage the available workspaces in the current working directory.
- Use the terraform workspace select command to change the currently selected workspace. For a given working directory, you can only select one workspace can be at a time. Most Terraform commands only interact with the currently selected workspace. This includes provisioning and state manipulation.
- **When you provision infrastructure in each workspace, you usually need to manually specify different input variables to differentiate each collection.** For example, you might deploy test infrastructure to a different region.

You can create multiple working directories to maintain multiple instances of a configuration with completely separate state data. However, Terraform installs a separate cache of plugins and modules for each working directory, so maintaining multiple directories can waste bandwidth and disk space. This approach also requires extra tasks like updating configuration from version control for each directory separately and reinitializing each directory when you change the configuration. Workspaces are convenient because they let you create different sets of infrastructure with the same working copy of your configuration and the same plugin and module caches.

A common use for multiple workspaces is to create a parallel, distinct copy of a set of infrastructure to test a set of changes before modifying production infrastructure.



```
terraform {

  cloud {
    organization = "MTSL-Org"

    workspaces {
      name = "startws"
    }
  }

  required_providers {
```

Terraform Cloud

- Log-in to Terraform Cloud.
- Go to WORKSPACES > +NEW WORKSPACE to create a new one.
- Give WorkSpace Name & Description > CREATE WORKSPACE.
- Run $terraform login to create a token, copy the link in terminal and create a token and enter its value. Authentication to TF Cloud is done!
- Run $terraform apply and your TF State file is mapped to your workspace under certain organization in Terraform Cloud.
- Delete TF State fil & TF State Lock File from your local console.
- GO to VARIABLES in TF Cloud toad sensitive vars.

https://www.terraform.io/docs/cloud/migrate/index.html - step-5-edit-the-backend-configuration

https://registry.terraform.io/providers/hashicorp/aws/latest/docs - environment-variables