Study Guide - Terraform Associate Certification.

▼ Author

👩‍🔧 Created by: Shiva Kumar B

▼ Let’s connect here

LinkedIn | GitHub

> A HashiCorp Certified Terraform Associate | Google Cloud Certified Leader | counting…

> Feel free to refer this for your prep & practice.

▼ Infrastructure as Code in a Private or Public Cloud

- As companies move their operations to the cloud they tend to manage their cloud infrastructure the same way they managed their on-premise physical hardware.
- by logging into their virtual infrastructure’s web interface, or directly onto a system and applying changes via GUI or CLI. These users haven’t adopted the use of infrastructure as code (IaC).
- Manual Configuring Infrastructure leads to,
  - Its easy to mis-configure a service through human error.

- Its hard to manage the expected state of configuration for compliance.
- Hard to transfer configuration knowledge to other team member.

▼ **What is IaC?**

▼ It is infrastructure (CPUs, memory, disk, firewalls, etc.) defined as code within definition files.

- You (author) write a script to `automate` `creating, updating` `&` `destroying` cloud infrastructure.
- IaC is `blueprint` of the Cloud Infrastructure.
- IaC allows you to easily share, version or inventory your cloud infrastructure.

▼ **IaC and the Infrastructure Lifecycle**

- IaC can be applied throughout the lifecycle, both on the initial build, as well as throughout the life of the infrastructure.
- Day 0-2 is a simplified way to describe phases of an infrastructure lifecycle.
- Day 0 —→ Plan & Design.
- Day 1 —→ Develop & Iterate.
- Day 2 —→ Go LIVE & Maintain.

> 📢 here days are not a 24hrs & it is just a broad way of defining where a infrastructure project would be.

▼ **Infrastructure Lifecycle Advantages**

IaC enhances the Infrastructure Lifecycle by,

- Reliability —→ IaC makes changes idempotent, consistent, repeatable, & predictable.
  - `Idempotent —→ The idempotent characteristic provided by IaC tools ensures that, even if the same code is applied multiple times, the result remains the same.`
- Manageability
  - Enable mutation via code.
  - revised, with minimal changes.
- Sensibility —→ avoid financial & reputational losses to even loss of life when considering govt. & military dependencies on infrastructure.

▼ **Provisioning vs Deployment vs Orchestration**

▼ Provisioning

- To prepare a server with system, data & software, & make it ready for network operation.
- Using Configuration management tools like puppet, chef, Ansible, BashScripts, PowerShell or Cloud-init you can provision a server.
- When you launch a cloud service & configure it as Provisioning.

▼ Deployment

- Deployment is the act of delivering a version of your application to run a provisioned server.
- Deployment could be performed via AWS cloud pipeline, Harness, Jenkins, Github Actions, Circle CI, Concourse CI etc..

▼ Orchestration

- It is the act of coordinating multiple systems or servers.
- It is a common term when working with microservices, containers & k8s.
- Orchestration could be Kubernetes, Salt, Fabric.

▼ Intro to Terraform

▼ What is Terraform

- HashiCorp Terraform is an infrastructure as code tool that lets you define both cloud and on-prem resources in human-readable configuration files that you can version, reuse, and share.
- You can then use a consistent workflow to provision and manage all of your infrastructure throughout its lifecycle.
- Terraform can manage low-level components like compute, storage, and networking resources, as well as high-level components like DNS entries and SaaS features.

▼ How does Terraform work?

▼ The core Terraform workflow consists of three stages:

▼ Write:

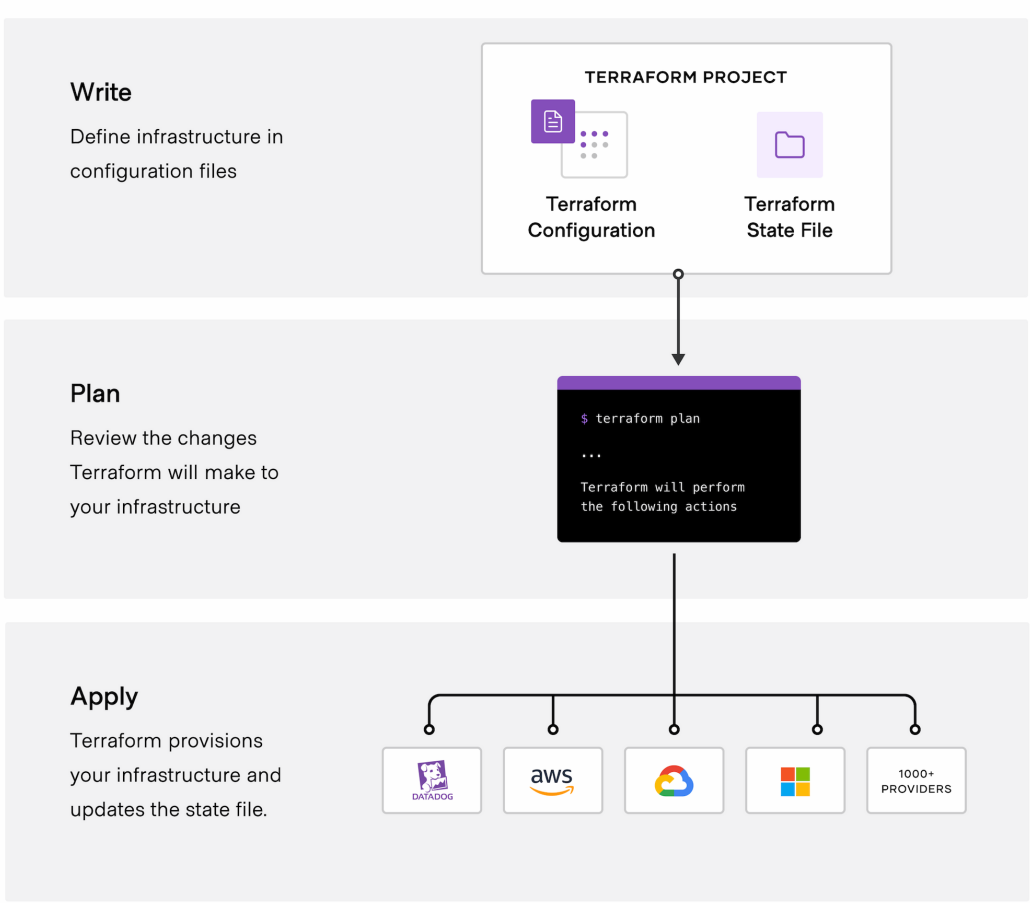
- You define resources, which may be across multiple cloud providers and services.
- For example, you might create a configuration to deploy an application on virtual machines in a Virtual Private Cloud (VPC) network with security groups and a load balancer.

▼ Plan:

- Terraform creates an execution plan describing the infrastructure it will create, update, or destroy based on the existing infrastructure and your configuration.

▼ Apply:

- On approval, Terraform performs the proposed operations in the correct order, respecting any resource dependencies.
- For example, if you update the properties of a VPC and change the number of virtual machines in that VPC, Terraform will recreate the VPC before scaling the virtual machines.



▼ **How can Terraform build infrastructure so efficiently?**

- Terraform builds a graph of all your resources, and parallelizes the creation and modification of any non-dependent resources.
- Because of this, Terraform builds infrastructure as efficiently as possible, and operators get insight into dependencies in their infrastructure.

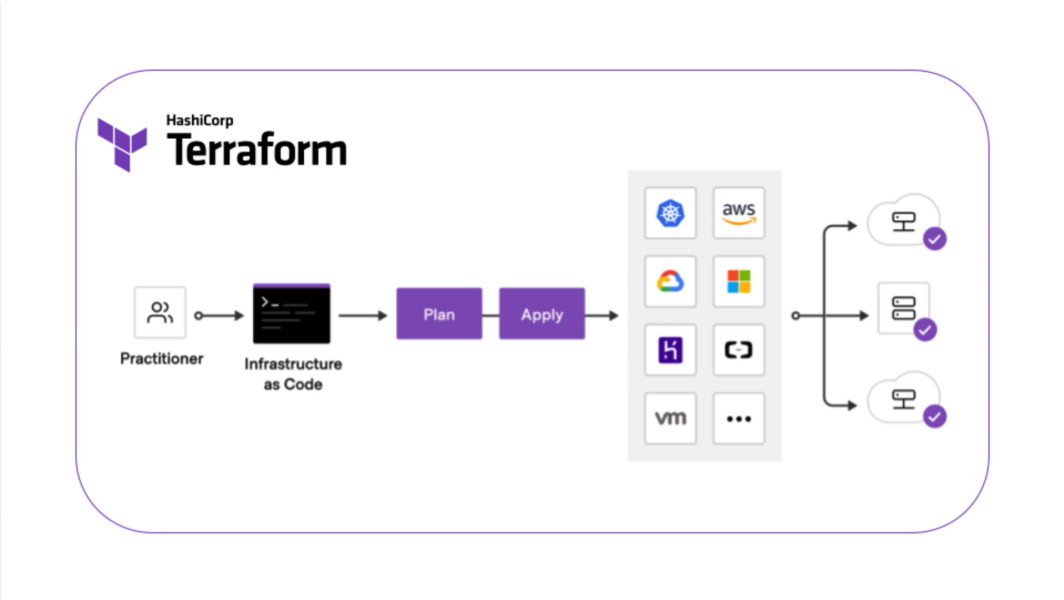
▼ **Why Terraform (Advantages)?**

1. **Declarative language:** Terraform uses high-level, declarative language to describe the desired state of your infrastructure. This makes it easier to understand and manage complex infrastructure deployments.
2. **Infrastructure as code:** Terraform treats infrastructure as code, which means that you can version control your infrastructure and collaborate with others more easily.
3. **Provider agnostic**: Terraform supports many popular cloud providers, as well as on-premises and other infrastructure resources. This means that you can use Terraform to manage a diverse range of infrastructure resources.
    a. It simplifies management and orchestration, helping operators build large-scale multi-cloud infrastructures.
4. **State management**: Terraform maintains a state file that describes the current state of your infrastructure. This state file is used to determine what changes need to be made in order to achieve the desired state.

5. **Modularity**: Terraform allows you to modularize your infrastructure, making it easier to manage complex deployments. You can use Terraform modules to encapsulate reusable components of your infrastructure and make it easier to manage.
6. **Idempotent operations**: Terraform only makes changes to your infrastructure if the desired state has changed. This means that you can safely run Terraform multiple times without unintended side effects.
7. **Testing**: Terraform provides a way to test your infrastructure changes before applying them to your actual infrastructure. This makes it easier to catch and fix issues before they affect production.

▼ **To Deploy Infra with Terraform**

- **Scope**: Identify the Infrastructure for your project
- **Author**: Write config. to define your Infrastructure
- **Initialize**: Install the required Terraform providers(plugins)
- **Plan**: Preview the changes that Terraform will make
- **Apply**: Make the changes to your Infrastructure



▼ **Terraform Cloud**

Terraform Cloud is a Software as a Service (SaaS) offering for:

- Remote Control Storage.
- Version Control integrations.
- Flexible workflows.
- Collaboration on Infrastructure changes in a single unified web portal.

> 💡 The underlying software for Terraform Cloud & Terraform Enterprise is known as "`Terraform Platform`"

▼ Use cases

▼ Common Use cases

- Provisioning and managing cloud infrastructure
- Automating data center deployments
- Multi-cloud deployments
- Continuous integration and delivery (CI/CD) pipelines
- Disaster recovery planning
- Security compliance
- Managed service deployment
- Automating the deployment of Kubernetes clusters
- Automating the deployment of network infrastructure
- Automating the deployment of application stacks
- Automating the deployment of serverless applications
- Automating the deployment of storage systems

- **IaC for Exotic Providers.**
- **Multi-tier Applications.**
- **Disposable Environments.**
- **Recourse Scheduler.**

▼ Multi-Cloud Deployment

- Provisioning your infrastructure into multiple cloud providers to increase the fault-tolerance of your applications.

▼ How multi-cloud deployment is useful?

- By using only a single region or cloud provider, fault tolerance is limited by the availability of that provider.
- Having a multi-cloud deployment allows for a more graceful recovery of the loss of a region or entire provider.

▼ Application Infrastructure Deployment, Scaling, and Monitoring Tools

- You can use Terraform to efficiently deploy, release, scale, and monitor infrastructure for multi-tier applications.
- N-tier application architecture lets you scale application components independently and provides a separation of concerns.

- An application could consist of a pool of web servers that use a database tier, with additional tiers for API servers, caching servers, and routing meshes.
- Terraform allows you to manage the resources in each tier together, and automatically handles dependencies between tiers. For example, Terraform will deploy a database tier before provisioning the web servers that depend on it.

▼ **Self-Service Clusters**

- A self-service cluster in Terraform refers to an infrastructure setup that enables users to provision and manage their own clusters, rather than relying on a centralized IT team.
- This type of setup is often used in organizations that have multiple teams or departments that need to run their own applications and services, and want to have more control over their infrastructure.

▼ **Software Defined Networking**

- Terraform can interact with Software Defined Networks (SDNs) to automatically configure the network according to the needs of the applications running in it.
- This lets you move from a ticket-based workflow to an automated one, reducing deployment times.

▼ **Kubernetes**

- Kubernetes is an open-source workload scheduler for containerized applications. Terraform lets you both deploy a Kubernetes cluster and manage its resources (e.g., pods, deployments, services, etc.).
- making it easier to provision and manage large-scale, multi-node Kubernetes deployments.

- **Policy Compliance and Management**
- **PaaS Application Setup**

▼ Terraform Basics

▼ Terraform is logically split into two main parts

- Terraform Core
  - uses remote procedural calls (RPC) to communicate with Terraform Plugins.
- Terraform Plugins
  - expose an implementation for specific service or provisioner.

- Terraform Core is a statistically-compiled binary written in the Go lang.
- reference
  - https://www.terraform-best-practices.com/

▼ HashiCorp Configuration Language (HCL)

- HCL is an open-source toolkit for creating `structured configuration language` that are both human & machine friendly, for use with command line tools.

▼ In use by

- terraform language (.tf)
- packer template (.pkr.hcl)
- vault policies (no extension)
- boundary controllers & workers (.hcl)
- Consul Configuration (.hcl)
- Waypoint Application Configuration (.hcl)
- Nomad Job Specifications (.nomad)
- Shipyard Blueprint (.hcl)

▼ Purpose of 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.
- State is a necessary requirement for Terraform to function.

▼ Mapping to the Real World

- Terraform requires some sort of database to map Terraform config to the real world because you can't find the same functionality in every cloud provider. You need to have some kind of mechanism to be cloud-agnostic.

▼ Metadata

- Terraform must also track metadata such as resource dependencies, pointer to the provider configuration that was most recently used with the resource in situations where multiple aliased providers are present.

▼ Performance

- In addition to basic mapping, Terraform stores a cache of the attribute values for all resources in the state. This is the most optional feature of Terraform state and is done only as a performance improvement.
- When running a `terraform plan`, Terraform must know the current state of resources in order to effectively determine the changes that it needs to make to reach the desired configuration.
- For ***small infrastructures***, Terraform can query your providers and sync the latest attributes from all your resources. This is the default behavior of Terraform: for every plan and apply, Terraform will sync all resources in your state.
- For ***larger infrastructures***, querying every resource is too slow. Many cloud providers do not provide APIs to query multiple resources at once, and the round trip time for each resource is hundreds of milliseconds. On top of this, cloud providers almost always have API rate limiting so Terraform can only request a certain number of resources in a period of time. Larger users of Terraform make heavy use of the `-refresh=false` flag as well as the `-target` flag in order to work around this. In these scenarios, the cached state is treated as the record of truth.

▼ **Syncing**

- In the default configuration, Terraform stores the state in a file in the current working directory where Terraform was run.
- when using Terraform in a team it is important for everyone to be working with the same state so that operations will be applied to the same remote objects.
- Remote State is the recommended solution to this problem. With a fully-featured state backend.
- Terraform can use remote locking as a measure to avoid two or more different users accidentally running Terraform at the same time, and thus ensure that each Terraform run begins with the most recent updated state.

▼ Terraform Settings

- The special `terraform` configuration block type is used to configure some behaviors of Terraform itself, such as requiring a minimum Terraform version to apply your configuration, etc.

▼ **Only constants are allowed inside the terraform block. Is this correct?**

- `Yes`
- Within a terraform block, only constant values can be used; arguments may not refer to named objects such as resources, input variables, etc, and may not use any of the Terraform language built-in functions.

▼ In terraform settings (block type), we can specify

▼ required_version

- the expected terraform version to be installed or use.

▼ required_providers

- the providers that will pull during `terraform init`.

▼ experiments

- experimental language features, that community can try & provide feedback.
- not recommended to use in the production environment.

```
terraform {
  experiments = [example]
}
```

▼ provider_meta

- module-specific inf. for providers.
- primarily intended for modules distributed by the same vendor as the associated provider.

```
terraform {
  provider_meta "my-provider" {
    hello = "world"
  }
}
```

▼ **Terraform Block Syntax**

```
terraform {
  # ...
  # other settings to configure
  # ...
}
```

- Terraform settings are gathered together into `terraform` blocks.
- Each `terraform` block can contain a number of settings related to Terraform's behavior.
- Within a `terraform` block, only constant values can be used, arguments may not refer to named objects such as resources, input variables, etc.. & may not use any of the terraform language built-in functions.

▼ **terraform.tf**

- *`terraform.tf` file* is used to configure the terraform settings, required versions, required providers & if wanted to use terraform cloud. providing cloud block will configure automatically to use terraform cloud inside the terraform block.

```
terraform {
 /*Uncomment this block to use Terraform Cloud for this tutorial
  cloud {
    organization = "organization-name"
    workspaces {
      name = "learn-terraform-provider-versioning"
    }
  }
*/

  required_providers {
    random = {
      source  = "hashicorp/random"
      version = "3.1.0"
    }

    aws = {
      source  = "hashicorp/aws"
      version = ">= 2.0.0"
    }
  }

  required_version = ">= 1.1"
}
```

▼ The various options supported within a `terraform` block are described as follows:

▼ **Configuring Terraform Cloud**

```
terraform {
  cloud {
    organization = "example_corp"
    ## Required for Terraform Enterprise; Defaults to app.terraform.io for Terraform Cloud
    hostname = "app.terraform.io"

    workspaces {
```

```
      tags = ["app"]
    }
  }
}
```

- The main module of a Terraform configuration can integrate with Terraform Cloud to enable its CLI-driven run workflow & only need to configure these settings to use Terraform CLI to interact with Terraform Cloud.
- To configure the Terraform Cloud CLI integration, add a nested `cloud` block within the `terraform` block.
- One cannot use the CLI integration and a state backend in the same configuration.

▼ **Configuring a Terraform Backend**

```
terraform {
  backend "remote" {
    organization = "example_corp"

    workspaces {
      name = "my-app-prod"
    }
  }
}
```

- A backend defines where Terraform stores its state data files.
- Available Backends are `local`, `remote`, `azurerm`, `consul`, `cos`, `gcs`, `HTTP`, `kubernetes`, `oss`, `pg`, `s3`.
- The `artifactory`, `etcd`, `etcdv3`, `manta`, and `swift` backends are removed in Terraform v1.3.
- By default, Terraform uses a backend called `local`, which stores the state as a local file on disk.
- No need of configuring backend when using Terraform cloud because Terraform cloud automatically manages state in the workspaces associated with config.
- If Terraform configuration includes `cloud block` then it cannot include `backend block`.

▼ **Limitations on backend configuration.**

- A configuration can only provide one backend block.
- A backend block cannot refer to named values (like input variables, locals, or data source attributes).

▼ **Credentials and Sensitive Data**

> ⛔ **Warning:** We recommend using environment variables to supply credentials and other sensitive data. If you use `-backend-config` or hardcode these values directly in your configuration, Terraform will include these values in both the `.terraform` subdirectory and in plan files. This can leak sensitive credentials.

- Backend stores state in a remote service, which allows more people to collaborate to work together, but state data contains extremely sensitive data.
- All plan files capture the information in `.terraform/terraform.tfstate` at the time the plan was created. which ensures TF is applying the plan to correct set of Infrastructure.
- Use environment variables to pass credentials when you need to use different values between the plan and apply steps.

▼ **Default Backend**

- If a configuration includes no backend block, Terraform defaults to using the `local` backend, which stores state as a plain file in the current working directory.

▼ **Initialization**

- When changing a backend config. one must run `terraform init` again to validate & configure the backend before applying plan or state operations.
- After initialize, Terraform creates a `.terraform/` directory locally. This directory contains the most recent backend configuration, including any authentication parameters you provided to the Terraform CLI. Do not check this directory into Git, as it may contain sensitive credentials for the remote backend.
- Changing backends, TF gives the option to migrate the state to the new backend to adopt backends without losing any existing state.

▼ **Partial Configuration**

- Not specifying every required argument in the backend block, When some or all of the arguments are omitted, configuring remaining arguments is known as ***partial configuration***.
- There are several ways to supply the remaining arguments:
  - **File**
  - **Command-line key/value pairs**
  - **Interactively**

▼ **Changing Configurations**

- Backend can be changed at anytime, also able to change both configuration itself & type of backend.
- Terraform automatically detects any changes made in the config files & request re-initialization process.
- As part of reinitialize process, Terraform will prompt to ask to migrate the existing state to the new config., This allows you to easily switch from one backend to another.
- If reconfiguring the same backend, Terraform will still ask if you want to migrate your state. You can respond "no" in this scenario.

▼ **Unconfiguring a Backend**

- If you no longer want to use any backend, you can simply remove the configuration from the file. Terraform will detect this like any other change and prompt you to reinitialize.
- As part of the reinitialization, Terraform will ask to migrate your state back down to normal local state. Once this is complete then Terraform is back to behaving as it does by default.

▼ **Specifying a Required Terraform Version**

👉 required_version = "~> 1.3.1”

- The required_version setting accepts a version constraint string, which specifies which versions of Terraform can be used with the configuration.
- If the running version (TF CLI) doesn’t match the config., Terraform will prompt an error & exit the process for provisioning/modifying the infra-services.
- When using `child modules` each module can specify its own version requirements. The requirements of all modules in the tree must be satisfied.
- Using TF version constraints in a collaborative Env., ensures that everyone using specific TF version or min version, this helps the TF version behavior as expected by configuration.
- The `required_version` setting applies only to the version of Terraform CLI. Terraform's resource types are implemented by provider plugins & independent of TF CLI.

▼ **Specifying Provider Requirements**

```
terraform {
  required_providers {
    aws = {
      version = ">= 2.7.0"
      source = "hashicorp/aws"
    }
  }
}
```

- The `required_providers` block specifies all of the providers required by the current module, mapping each `local provider name` to a `source address` and a `version constraint`.

▼ Providers in Terraform

▼ Terraform Providers

- Providers are Terraform Plugins that allow us to interactive with
  - Cloud Service Providers (CSP) eg. AWS, Azure, GCP.
  - SaaS Providers eg. Github, Angolia, Stripe.
  - Other APIs eg. K8s, Postgres etc..
- A provider is responsible for understanding API interactions and exposing resources.
- Providers are required for terraform configuration file to work.

- Providers come in three tiers
  - Official.
  - Verified.
  - Community.
- Providers are distributed separately from terraform & the plugins must be downloaded before use.
  - `terraform init` will download the necessary provider plugins listed in Terraform Config file.

▼ **What are the meta-arguments that are defined by Terraform itself and available for all `provider` blocks?**

- **version:** Constraining the allowed provider versions.
- **alias:** using the same provider with different configurations for different resources.

▼ **How do you upgrade to the latest acceptable version of the provider?**

- `terraform init --upgrade`
- It upgrade to the latest acceptable version of each provider this command also upgrades to the latest versions of all Terraform modules.

▼ **How many ways you can configure provider versions?**

```
1. With required_providers blocks under terraform blockterraform {
  required_providers {
    aws= "~> 1.0"
  }
}


2. Provider version constraints can also be specified using a version argument within a provider
block
provider {
  version= "1.0"
}
```

▼ **Terraform Registry**

- It is a website portal to browse, download or publish available providers or modules.
- Provider
  - A provider is a plugin that is a mapping to a CSPs API.
- Module
  - A module is a group of configuration files that provide common config functionality.
  - Enforces best practices.
  - reduce the amount of code.
  - reduce time to develop the script.
- `terraform provider` is a terraform command to view what providers we have installed for our project.

▼ **Provider Configuration**

- Providers allow Terraform to interact with cloud providers, SaaS providers, and other APIs.
- Some providers require you to configure them with endpoint URLs, cloud regions, or other settings before Terraform can use them.
- Provider configurations belong in the root module of a Terraform configuration.
- A provider configuration is created using a `provider` block

```
provider "google" {
  project = "acme-app"
  region  = "us-central1"
}
```

- The name given in the block header (`"google"` in this example) is the local name of the provider to configure. This provider should already be included in a `required_providers` block.
- The body of the block (between `{` and `}`) contains configuration arguments for the provider.
- In this example both `project` and `region` are specific to the `google` provider.
- One can use expressions in the values of these configuration arguments, but can only reference values that are known before the configuration is applied. This means you can safely reference input variables, but not attributes exported by resources (with an exception for resource arguments that are specified directly in the configuration).
- Some providers can use shell environment variables (or other alternate sources, like VM instance profiles) as values for some of their arguments; when available, we recommend using this as a way to keep credentials out of your version-controlled Terraform code.

▼ **Multiple Provider Configuration (Alternative Providers)**

- The primary reason for this is to support multiple regions for a cloud platform; other examples include targeting multiple Docker hosts, multiple Consul hosts, etc.

▼ Set an alternative provider

- using `alias`, it is a meta argument provided by terraform for all the providers.

```
provider "aws" {
  alias = "west"
  region  = "us-west-2"
}
```

▼ How to reference an alias provider

- using `<PROVIDER NAME>.<ALIAS>` eg. aws.west from previous example

```
resource "aws_instance" "web" {
  provider = aws.west
}
```

▼ How to set alias provider for parent module

- using `configuration_aliases` meta-argument in terraform setting.

```
terraform {
  required_providers {
    mycloud = {
      source  = "mycorp/mycloud"
      version = "~> 1.0"
      configuration_aliases = [ mycloud.alternate ]
    }
  }
}
```

▼ How to set alias provider for child module

```
module "aws_vpc" {
  source = "./aws_vpc"
  providers = {
    aws =aws.west
  }
}
```

▼ **What is Provider Plugin Cache?**

- By default, terraform init downloads plugins into a subdirectory of the working directory so that each working directory is self-contained. As a consequence, if you have multiple configurations that use the same provider then a separate copy of its plugin will be downloaded for each configuration.
- Given that provider plugins can be quite large (on the order of hundreds of megabytes), this default behavior can be inconvenient for those with slow or metered Internet connections.
- Therefore Terraform optionally allows the use of a local directory as a shared plugin cache, which then allows each distinct plugin binary to be downloaded only once.

▼ **How do you enable Provider Plugin Cache?**

- To enable the plugin cache, use the plugin_cache_dir setting in the CLI configuration file.
- `plugin_cache_dir = "$HOME/.terraform.d/plugin-cache"`
- Alternatively, the **TF_PLUGIN_CACHE_DIR** environment variable can be used to enable caching or to override an existing cache directory within a particular shell session.
- Once the task is completed, It is USER responsibility to clean up or delete a a plugin from the plugin cache once it's been placed there.Over time, as plugins are upgraded, the cache directory may grow to contain several unused versions which must be manually deleted.

▼ How to Use Providers

> To use resources from a given provider, you need to include some information about it in your configuration.

- Provider Requirements documents how to declare providers so Terraform can install them.

- Provider Configuration documents how to configure settings for providers.
- Dependency Lock File documents an additional HCL file that can be included with a configuration, which tells Terraform to always use a specific set of provider versions.

▼ Provider Requirements

- Terraform relies on plugins called `"providers"` to interact with remote systems.
- Terraform configurations must declare which providers they require, so that Terraform can install and use them.
- Additionally, some providers require configuration (like endpoint URLs or cloud regions) before they can be used.

▼ Requiring Providers

- Each Terraform module must declare which providers it requires, so that Terraform can install and use them.
- Provider requirements are declared in a `required_providers` block.
- A provider requirement consists of a local name, a source location, and a version constraint, ex as shown below

```
terraform {
  required_providers {
    mycloud = {
      source  = "mycorp/mycloud"
      version = "~> 1.0"
    }
  }
}
```

- The `required_providers` block must be nested inside the top-level `terraform` block (which can also contain other settings).

▼ Each argument in the `required_providers` block enables one provider. The key determines the provider's local name (its unique identifier within this module), and the value is an object with the following elements:

- `source` - the global source address for the provider you intend to use, such as `hashicorp/aws`.
- `version` - a version constraint specifying which subset of available provider versions the module is compatible with.

> The `name = { source, version }` syntax for `required_providers` was added in **Terraform v0.13**.

> Previous versions of Terraform used a version constraint string instead of an object (like `mycloud = "~> 1.0"`), and had no way to specify provider source addresses.

▼ **Names and Addresses**

▼ Each provider has two identifiers:

- A unique *source address*, which is only used when requiring a provider.
- A *local name*, which is used everywhere else in a Terraform module.

> 📌 Prior to Terraform 0.13, providers only had local names, since Terraform could only automatically download providers distributed by HashiCorp.

▼ **Local Names**

- Local names are module-specific, and are assigned when requiring a provider.
- Local names must be unique per-module.
- Outside of the `required_providers` block, Terraform configurations always refer to providers by their local names. For example, the following configuration declares `mycloud` as the local name for `mycorp/mycloud`, then uses that local name when configuring the provider

```
terraform {
  required_providers {
    mycloud = {
      source  = "mycorp/mycloud"
      version = "~> 1.0"
    }
  }
}

provider "mycloud" {
  # ...
}
```

> 💡 For example, resources from `hashicorp/aws` all begin with `aws`, like `aws_instance` or `aws_security_group`.

▼ **Source Addresses**

- A provider's source address is its global identifier. It also specifies the primary location where Terraform can download it.

▼ Source addresses consist of three parts delimited by slashes (`/`), as follows:

> 📌 [<HOSTNAME>/]<NAMESPACE>/<TYPE>

- **Hostname** (optional): The hostname of the Terraform registry that distributes the provider. If omitted, this defaults to `registry.terraform.io`, this is the hostname of the public Terraform Registry.

- **Namespace:** An organizational namespace within the specified registry. For the public Terraform Registry and for Terraform Cloud's private registry, this represents the organization that publishes the provider. This field may have other meanings for other registry hosts.
- **Type:** A short name for the platform or system the ***provider manages.*** Must be unique within a particular namespace on a particular registry host.

- The source address with all three components given explicitly is called the provider's `fully-qualified address`.
- You will see fully-qualified address in various outputs, like error messages, but in most cases a simplified display version is used. This display version omits the source host when it is the public registry, so you may see the shortened version `"hashicorp/random"` instead of `"registry.terraform.io/hashicorp/random"`.

▼ **Handling Local Name Conflicts**

- Whenever possible, we recommend using a provider's preferred *local name*, which is usually the same as the *"type" portion of its source address*.
- However, it's sometimes necessary to use two providers with the same preferred local name in the same module, usually when the providers are named after a generic infrastructure type. Terraform requires unique local names for each provider in a module, so you'll need to use a non-preferred name for at least one of them.
- When this happens, we recommend combining each provider's namespace with its type name to produce compound local names with a dash:

```
terraform {
  required_providers {
    # In the rare situation of using two providers that
    # have the same type name -- "http" in this example --
    # use a compound local name to distinguish them.
    hashicorp-http = {
      source  = "hashicorp/http"
      version = "~> 2.0"
    }
    mycorp-http = {
      source  = "mycorp/http"
      version = "~> 1.0"
    }
  }
}

# References to these providers elsewhere in the
# module will use these compound local names.
provider "mycorp-http" {
  # ...
}

data "http" "example" {
  provider = hashicorp-http
  #...
}
```

▼ **Version Constraints**

- Each provider plugin has its own set of available versions, allowing the functionality of the provider to evolve over time.
- Each provider dependency you declare should have a version constraint given in the `version` argument so Terraform can select a single version per provider that all modules are compatible with.
- The `version` argument is optional; if omitted, Terraform will accept any version of the provider as compatible.
- It is strongly recommended specifying a version constraint for every provider your module depends on.
- To ensure Terraform always installs the same provider versions for a given configuration, you can use Terraform CLI to create a dependency lock file and commit it to version control along with your configuration.
- If a lock file is present, Terraform Cloud, CLI, and Enterprise will all obey it when installing providers.
- **Hands-on:** Try the Lock and Upgrade Provider Versions tutorial.

▼ **Best Practices for Provider Versions**

▼ Each module should at least declare the minimum provider version it is known to work with, using the `>=` version constraint syntax

```
terraform {
  required_providers {
    mycloud = {
      source  = "hashicorp/aws"
      version = ">= 1.0"
    }
  }
}
```

▼ The `~>` operator is a convenient shorthand for allowing the rightmost component of a version to increment.

- The following example uses the operator to allow only patch releases within a specific minor release**.**

```
terraform {
  required_providers {
    mycloud = {
      source  = "hashicorp/aws"
      version = "~> 1.0.4"
    }
  }
}
```

- Do not use `~>` for modules you intend to reuse across many configurations, even if you know the module isn't compatible with certain newer versions.
- Doing so can sometimes prevent errors, but more often it forces users of the module to update many modules simultaneously when performing routine upgrades.

- Specify a minimum version, document any known incompatibilities, and let the root module manage the maximum version.

▼ **Built-in Providers**

- Most Terraform providers are distributed separately as plugins, but there is one provider that is built into Terraform itself.
- This provider enables the the `terraform_remote_state` data source.
- Because this provider is built into Terraform, you don't need to declare it in the `required_providers` block in order to use its features.
- for consistency it *does* have a special provider source address, which is `terraform.io/builtin/terraform`.
- This address may sometimes appear in Terraform's error messages and other output in order to unambiguously refer to the built-in provider, as opposed to a hypothetical third-party provider with the type name "terraform".

▼ **In-house Providers**

- Anyone can develop and distribute their own Terraform providers.
- Some organizations develop their own providers to configure proprietary systems, and wish to use these providers from Terraform without publishing them on the public Terraform Registry.
- One option for distributing such a provider is to run an in-house *private* registry, by implementing the provider registry protocol.

> 📌 Explicit provider source addresses were introduced with Terraform v0.13, so the full provider requirements syntax is not supported by Terraform v0.12.

▼ Provisioners

> ⚠️ **Important:**
> Use provisioners as a last resort. There are better alternatives for most situations.

- Provisioners in Terraform are used to execute scripts on the local or remote machine in order to provision the server. Examples of provisioners include installing packages, creating users, uploading files, and running services.
- Terraform allows you to work with different provisions like Cloud-Init, Packer etc.
- Provisioners are only run when a resource is *created or destroyed*, Provisioners that are run while destroying are Destroy provisioners.

▼ Let's look at an example of using a local-exec provisioner.

```
# example-1 :-
```

```
resource "aws_instance" "web_server" {
  ami           = "ami-example"
  instance_type = "t2.micro"

  # Create an Nginx web server
  provisioner "local-exec" {
    command = "bash /path/to/deploy_nginx.sh"
  }
}
```

▼ **local-exec Provisioner**

- `local-exec` provisioner allows you to run scripts or commands on the local machine where Terraform is being executed.
- The machine that is executing `terraform apply` is where the command will execute.

▼ Local Env could be

- Local Machine —> laptop / workstation.
- Build Server —> GCP Cloud Build, AWS Cloud Build, Jenkins.
- Terraform Cloud Run Env —> single-use Linux virtual machine.

▼ outputs vs local-exec

- Terraform outputs allows to output results after running Terraform apply.
- local exec allows you run any arbitrary commands on your local machine.
- commonly used to trigger Config. Management like Ansible, CHef, Puppet.

▼ arguments & examples

- Arguments

▼ command (required)

- the command you want to execute.

▼ working_dir

- where the command will executed
- ex. /user/home/ubuntu/project

▼ interpreter

- The entry point for the command.
- what local program will run the command.
- eg. Bash, Ruby, AWS CLI, PowerShell etc..

▼ Environment

- define & store environment values.

- examples

▼ ex1

```
resource "aws_instance" "web" {
  # ...

  provisioner "local-exec" {
    command = "echo ${self.private_ip} >> private_ips.txt"
  }
}
```

▼ interpreter example

```
resource "null_resource" "example2" {
  provisioner "local-exec" {
    command = "Get-Date > completed.txt"
    interpreter = ["PowerShell", "-Command"]
  }
}
```

▼ env. values example

```
resource "null_resource" "example2" {
  provisioner "local-exec" {
    command = "echo $KEY $SECRET >> credentials.yml"

    environment = {
      KEY = "your env key value."
      SECRET = "your env secret value."
    }
  }
}
```

▼ **remote-exec Provisioner**

- The `remote-exec` provisioner allows you to execute commands on a target resource after resource is provisioned.
- `remote-exec` is useful for provisioning a VM with a simple set of commands.
- for more complex tasks it is recommended to use Cloud-Init, & strongly recommend in all cases to bake golden images via packer or EC2 image builder.

▼ The `remote-exec` provisioner requires a connection and supports both `ssh` and `winrm`.

```
resource "aws_instance" "web" {
  # ...

  # Establishes connection to be used by all
  # generic remote provisioners (i.e. file/remote-exec)
  connection {
    type     = "ssh"
    user     = "root"
    password = var.root_password
    host     = self.public_ip
  }

  provisioner "remote-exec" {
    inline = [
      "puppet apply",
      "consul join ${aws_instance.web.private_ip}",
```

```
    ]
  }
}
```

▼ remote-exec command has three different stages

- inline → list of commands strings.
- script → relative or absolute local script that will be copied to the remote resource & then executed.
- scripts → relative or absolute local script that will be copied to the remote resource & then executed in order.

- one can only choose one at a time.

▼ examples

```
// example 1

provisioner "remote-exec" {
    inline = [
      "chmod +x /tmp/script.sh",
      "/tmp/script.sh args",
    ]
  }
```

```
// example 2

provisioner "remote-exec" {
    scripts = [
      "/setup.sh",
      "/home/tmp/script.sh args",
    ]
  }
```

▼ **File Provisioner**

- The `file` provisioner copies files or directories from the machine running Terraform to the newly created resource.
- The `file` provisioner supports both `ssh` & `winrm` type connections.

▼ arguments & examples

- arguements

  ▼ source

  - the local file we want to upload to the remote machine.

  ▼ content

  - a file or folder

  ▼ Destination

  - where you want to upload file on the remote server.

- examples

```
resource "aws_instance" "web" {
  # ...

  # Copies the myapp.conf file to /etc/myapp.conf
  provisioner "file" {
    source      = "conf/myapp.conf"
    destination = "/etc/myapp.conf"
  }

  # Copies the string in content into /tmp/file.log
  provisioner "file" {
    content     = "ami used: ${self.ami}"
    destination = "/tmp/file.log"
  }

  # Copies the configs.d folder to /etc/configs.d
  provisioner "file" {
    source      = "conf/configs.d"
    destination = "/etc"
  }

  # Copies all files and folders in apps/app1 to D:/IIS/webapp1
  provisioner "file" {
    source      = "apps/app1/"
    destination = "D:/IIS/webapp1"
  }
}
```

▼ Connection Block

- A `connection` block tells a provisioner or resource, how to establish a connection.
- we can establish connection or connect to the remote machine using `SSH` & `WinRM`.
- A `connection block` nested in a `provisioner block` only affects that provisioner and overrides any resource-level connection settings.

```
// SSH connection
# Copies the file as the root user using SSH
provisioner "file" {
  source      = "conf/myapp.conf"
  destination = "/etc/myapp.conf"

  connection {
    type     = "ssh"
    user     = "root"
    password = "${var.root_password}"
    host     = "${var.host}"
  }
}

// WinRM connection
# Copies the file as the Administrator user using WinRM
provisioner "file" {
  source      = "conf/myapp.conf"
  destination = "C:/App/myapp.conf"

  connection {
    type     = "winrm"
    user     = "Administrator"
    password = "${var.admin_password}"
    host     = "${var.host}"
```

```
  }
}
```

- expressions can use the `self`-object, which represents the connection's parent resource and has all of that resource's attributes.
- but cannot refer to their parent resource by referring to a resource by name within its own block would create a dependency cycle.
- refer for more detail: https://developer.hashicorp.com/terraform/language/resources/provisioners/connection#connection-block

▼ **null resource**

- Provisioners without a Resource
- If you need to run provisioners that aren't directly associated with a specific resource, you can associate them with a `null_resource`.
- Instances of `null_resource` are treated like normal resources, but they don't do anything. Like with any other resource, you can configure provisioners & connection details on a `null_resource`.

▼ Variables & Data

▼ Input Variables

```
variable "region" {
  description = "AWS region."
  type = string
  value = "us-west-2"
}
```

- Input variables (aka terraform variables or variables) are parameters for terraform for terraform modules.
- Variables are defined via `variable` blocks.
- Variable declaration in either
  - The root modules.
  - The child modules.

▼ Variable arguments

- description → This specifies the variable documentation.
- type → specifies what value type is accepted for the variable.
- default → default value which makes the variable optional.
- validation → A block to define validation rules, usually in addition to type constraints.
- sensitive → Limits Terraform UI output when the configuration uses the variable.

▼ Variable Definition Files

- A variable definitions file allows you to set the values for multiple variables at once.
- Variable definition files are named `.tfvars` or `tfvars.json`
- By default, `terraform.tfvars` will be autoloaded when included in the root of the project directory.
- Variable definition files uses Terraform language.

▼ Variables via Environment Variables

- A variable value can be defined by Environment Variables.
- Variable starting with `TF_VAR<name>` will be read & loaded.

```
export TF_VAR_image_id=ami-3eu3911
```

▼ Loading Input Variables

▼ default autoloaded variables file

- terraform.tfvars
- when you created naming file with terraform.tfvars, it will be automatically loaded when running terraform apply.

▼ Additional variable files (not autoloaded)

- my_variables.tfvars.
- We can create additional variable files eg. dev.tfvars, prod.tfvars, etc.
- they will not be loaded (we need to specify this in command line).

▼ Additional variable files (autoloaded)

- my_variables.auto.tfvars.
- if we name your file with auto.tfvars, it will always be loaded.

▼ Specify variable file via CLI

- -var-file “dev.tfvars”
- can specify variablefile using this flag with terraform apply.

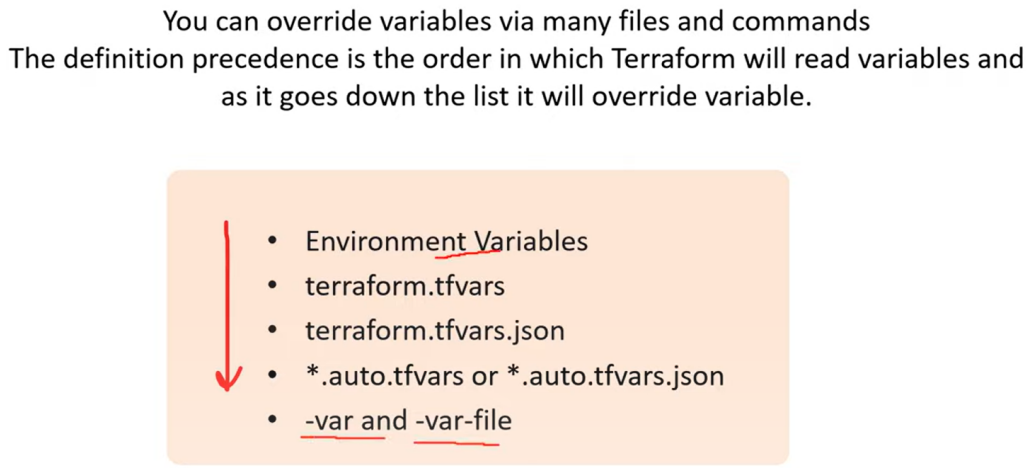
▼ Inline variable via CLI

- -var ec2_type=”t2.micro”
- Can specify variable inline via the command line for individual overrides.

▼ Env. variables

- TF_VAR_my_variable_name, TF_VAR_aws_access_key_id
- Terraform will watch for environment variables that begin with TF_VAR_ & apply those as variables.

▼ Variable Definition Precedence



▼ Output Values

- Output Values are computed values after a Terraform apply is performed.
- Outputs allows
  - to obtain information after resource provisioning eg. public_ip.
  - output a file of values for programmatic integration.
  - Cross- reference stacks via outputs in a state file via `terraform_remote_state`.

```
output "db_password" {
  value       = aws_db_instance.db.password
  description = "The password for logging in to the database."
  sensitive   = true
}
```

- value (required), description (Optional).
  - We can mark the output as sensitive so it does not show in the output of the terminal.
  - sensitive outputs will still be visible within the state file.

▼ terraform output commands

- To print all the outputs in the terminal.

```
terraform output
```

- Print a specific output value.

```
terraform output lb_url
```

- use -json flag to get output as json data.

```
terraform output -json
```

- use the -raw flag to preserve quotes for strings.

```
curl $(terraform output -raw lb_url)
```

▼ Local Values

▼ 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.

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

▼ Syntax

```
// static values
locals {
  service_name = "forum"
  owner        = "Community Team"
}

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

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

- Once a local value is declared, you can reference it in expressions as `local.<NAME>` .
- when declaring use plural “locals” & referencing, use singular “`local`” .
- Locals can help ***DRY*** ***up your code.***
  - It is best practice to use locals sparingly since it terraform is intended to be declarative & overuse of locals can make it difficult to determine what the code is doing.

▼ Data Sources

- *Data sources* allow Terraform to use information defined outside of Terraform, defined by another separate Terraform configuration, or modified by functions.

▼ We can specify what kind of external resource, we want to select.

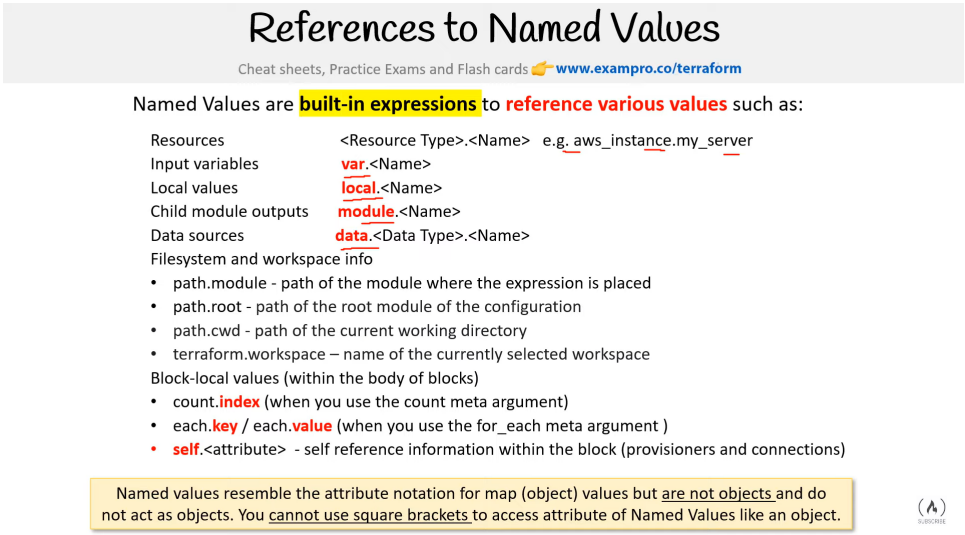
- using filters to narrow down the selection.

```
# Find the latest available AMI that is tagged with Component = web
data "aws_ami" "web" {
  filter {
    name   = "state"
    values = ["available"]
  }

  filter {
    name   = "tag:Component"
    values = ["web"]
  }

  most_recent = true
}
```

▼ using `data` to reference data sources.

```
resource "aws_instance" "web" {
  ami           = data.aws_ami.web.id
  instance_type = "t1.micro"
}
```

▼ References to Named Values



▼ Resource Meta Arguments

▼ `depends_on`

- The order in which resources are provisioned is important when resources depend on others before they are provisioned.
- Terraform implicitly can determine the order of provision for resources but there may be some cases where it cannot determine the correct order.
- `depends_on` allows you to explicitly specify a dependency of the resource.

```
resource "aws_iam_role" "example" {
  name = "example"
  role = aws_iam_role.example
}

resource "aws_instance" "example" {
  ami           = "ami-a1b2c3d4"
  instance_type = "t2.micro"

  depends_on = [
    aws_iam_role_policy.example
  ]
}
```

▼ `count`

- when managing a pool of objects eg. a fleet of VMs then we can use `count` to specify the number of instances we want.
- `count.index`
  - to get a current count value via `count.index`
  - this value starts with `0`.
- `count` can accept numeric expressions.
  - must be whole number.
  - number must be known before configuration.

A given resource or module block cannot use both `count` and `for_each`.

```
resource "aws_instance" "server" {
  count = 4 # create four similar EC2 instances

  ami           = "ami-a1b2c3d4"
  instance_type = "t2.micro"

  tags = {
    Name = "Server ${count.index}"
  }
}
```

▼ `for_each`

- `for_each` is similar to count for managing multiple related objects, but you can iterate over a map for more dynamic values.

▼ with a map

- `each.key` → print out the current key.
- `each.value` → print out the current value.

```
resource "azurerm_resource_group" "rg" {
for_each = {
a_group = "eastus"
```

```
another_group = "westus2"
}
name     = each.key
location = each.value
}
```

▼ with a list

- `each.key` → print out the current key.

```
resource "aws_iam_user" "the-accounts" {
  for_each = toset( ["Todd", "James", "Alice", "Dottie"] )
  name     = each.key
}
```

▼ `provider` & `alias`

```
# default configuration
provider "google" {
  region = "us-central1"
}

# alternate configuration, whose alias is "europe"
provider "google" {
  alias  = "europe"
  region = "europe-west1"
}

resource "google_compute_instance" "example" {
  # This "provider" meta-argument selects the google provider
  # configuration whose alias is "europe", rather than the
  # default configuration.
  provider = google.europe

  # ...
}
```

- If you need to override the default `provider` for a resource you can create *`alternative provider`* with an `alias`.
- referencing an `alias` in the resource configuration or object will change the default provider with alternative provider.

▼ Resource Behaviour & LifeCycle

▼ Resource Behaviour

- when you execute an execution order via `terraform apply` it will perform one of the following to a resouce.
  - create `+`
  - destroy `-`
  - update-in-place `~`
  - destroy & re-create `-/+`

▼ Resource LifeCycle

- Lifecycle block allows you to change what happens to resource eg. create, destroy, update.
- Lifecycle block are nested within resources.

```
resource "azurerm_resource_group" "example" {
  # ...

  lifecycle {
    create_before_destroy = true
  }
}
```

- `create_before_destroy` (bool) → when replacing a resource first create new resource before deleting it (the default is to destroy old first).
- `prevent_destroy` (bool) → ensures a resource is not destroyed.
- `ignore_changes` (list of attributes) → don't change the resource (create, update, destroy) the resource if a change occurs for the listed attributes.

▼ Terraform Expressions

- Expressions are used to refer to or compute values within the configuration.

▼ Types & Values

▼ primitive type

▼ string

- A double-quoted string can interpret escape sequences like /n/t /s etc.
- spl escape sequence like $${ (without beginning an interpolation sequence) or %%{ (without beginning a template directive sequence).

▼ Terraform also supports a *`heredoc`* style.

- heredoc is a UNIX style multi-line string.

```
// example 1
<<EOF
hello
world
EOF
```

```
// example 2
block {
  value = <<EOT
hello
world
EOT
}
```

- number

- bool

▼ no type

- `null`

```
var_name= null
```

- null represents absence or omission.
- when you want to use the underlying default of a provider's resource configuration option.

▼ complex/structural/collection types

- list(tuple)
- map(object)

▼ Strings Templates

▼ **String Interpolation** allows to evaluate an expression between the markers eg. `${…}` & converts it to a string.

```
"Hello ${var.name}"
```

▼ **String directive** allows to evaluate an conditional logic between markers eg. `%{…}`

```
"Hello, %{ if var.name != "" }${var.name}%{ else }unnamed%{ endif }!"
```

▼ **Whitespace Stripping** allows template directives to be formatted for readability without adding unwanted spaces and newlines to the result.

```
<<EOT
%{ for ip in aws_instance.example.*.private_ip ~}
server ${ip}
%{ endfor ~}
EOT
```

- We can use interpolation or directives within a `heredoc`.

▼ Conditional Expressions

- Terraform supports *`ternary if else`* conditions.
  - `condition? true_val : false_val`
  - `var.a != "" ? var.a : "default-a”`
- The return type for it the if & else must be the same type.
  - `var.example ? tostring(12) : "hello”`

▼ For Expressions

- For expressions allows to iterate over a complex type & apply transformations.

- A for expression can accept as input a list, a tuple, a map or an object.

▼ examples

▼ if `var.list` were a list of strings, then the following expression would produce a tuple of strings with all-uppercase letters.

```
[for s in var.list : upper(s)]
```

▼ for map, you can get: Key & Value.

```
[for k, v in var.map : length(k) + length(v)]
```

▼ for a list, you can get: Index.

```
[for i, v in var.list : "${i} is ${v}"]
```

▼ curly braces returns an object.

```
{for s in var.list : s => upper(s)}
```

▼ square braces returns a tuple.

```
[for s in var.list : s => upper(s)]
```

▼ **Filtering Elements**

- An `if statement` can be used in a for expression to filter/reduce the amount of elements returned.
- `[for s in var.list : upper(s) if s != ""]`

▼ **Implicit Element Ordering on Conversion (Element Ordering)**

- `for` expressions can convert from unordered types (maps, objects, sets) to ordered types (lists, tuples), Terraform must choose an implied ordering for the elements of an unordered collection.
- **For maps and objects**, Terraform sorts the elements by key or attribute name, using lexical sorting.
- **For sets of strings**, Terraform sorts the elements by their value, using lexical sorting.

▼ Splat Expressions

- A `splat expression` provides a `shorter` expression for `for expressions`.

▼ what is splat operator?

- A splat operator is represented by an asterisk `*` .
- it originates from ruby language.
- splat in terraform is used to roll up or soak-up a bunch of iterations in a `for expression` .

▼ for lists, sets, tuples

```
[for o in var.list : o.id]
[for o in var.list : o.interfaces[0].name]
```

- converted to using splat expressions.

```
var.list[*].id
var.list[*].interfaces[0].name
```

▼ Splat expression have a spl behavior when apply them to lists.

- If the value is anything other than a null value then the splat expression will transform it into a single-element list.
- If the value is a null value then the splat expression will return an empty tuple.

▼ Dynamic Blocks

- Dynamic Blocks allows to dynamically construct repeatable nested blocks.

```
locals {
  inbound_ports = [80, 443]
  outbound_ports = [443, 1433]
  }

# Security Groups
resource “aws_security_group” “sg-webserver” {
  vpc_id = aws_vpc.vpc.id
  name = “webserver”
  description = “Security Group for Web Servers”
  dynamic “ingress” {
    for_each = local.inbound_ports
    content {
      from_port = ingress.value
      to_port = ingress.value
      protocol = “tcp”
      cidr_blocks = [ “0.0.0.0/0” ]
    }
  }
  dynamic “egress” {
    for_each = local.outbound_ports
    content {
      from_port = egress.value
      to_port = egress.value
      protocol = “tcp”
      cidr_blocks = [ var.vpc-cidr ]
    }
  }
}
```

▼ Version Constraints

- Terraform uses semantic versioning for specifying Terraform Providers & Modules versions.

▼ `semantic versioning` is an open standard on how to define versioning for software management eg. `MAJOR.MINOR.PATCH`

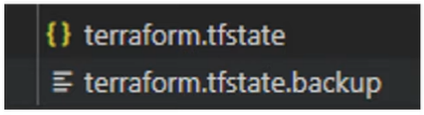
1. MAJOR version when you make incompatible API changes.
2. MINOR version when you add functionality in backwards compatible manner.
3. PATCH version when you make backwards compatible bug fixes.
   - Additional labels for pre-release & build meta-data are available as extensions to the **MAJOR.MINOR.PATCH** format.

- A `version constraint` is string containing one or more conditions, separated by commas.

▼ Terraform State Backups

All terraform state subcommands that *modify state* will write a backup file.
Read only commands will not modify state eg. list, show

Terraform will take the current state and store it in a file called **terraform.tfstate.backup**



Backups cannot be disabled. This is by design to enforce best-practice for recovery

To get rid of the backup file you need to manually delete the files

▼ Manage Resources in Terraform State

- After creating Infrastructure & State, we can examine state using `terraform.tfstate` file or with CLI using `terraform show` cmd (**SAFE MODE**).
- In `terraform.tfstate` state file, 1st block will have the `terraform setting` & following that resources where each resource will have `data as a mode or managed` & `type of resources`, `dependencies`, etc.

▼ **Remove the resource & import it using TF subcommands.**

- Assuming you already have a setup of AWS infrastructure (or any other).
- I have a setup with EC2 & Security group, here I will use security group as example.

▼ **Remove a resource from state**

- The `terraform state rm` subcommand removes specific resources from your state file.

- This does not remove the resource from your configuration or destroy the infrastructure itself.

```
terraform state rm aws_security_group.sg_8080
```

```
terraform state list
```

- The removed `security_group` resource does not exist in the state, but the resource still exists in your AWS account.
- Run `terraform import` to bring this security group back into your state file.
- Removing the security group from the state did not remove the output value with its ID, so we can use it for the import.

```
terraform import aws_security_group.sg_8080 $(terraform output -raw security_group)
```

⚠️ **Note:** If you were to apply the configuration before importing your resource back into state, Terraform would create a new resource, and your original infrastructure would keep running until you manually removed it.

▼ **Refresh modified infrastructure**

- The `terraform refresh` command updates the state file when physical resources change outside of the Terraform workflow.

⚠️ **Note:**
Terraform automatically performs a `refresh` during the `plan`, `apply`, and `destroy` operations.
All of these commands will reconcile state by default, and have the potential to modify your state file.

▼ **Replace a resource with CLI**

```
terraform plan -replace="aws_instance.example"
```

- Replacing a resource is useful in cases where a user manually changes a setting on a resource or when you need to update a provisioning script.
- Only Managed resources are used to replace using `-replace` flag in terraform apply
- This allows you to rebuild specific resources and avoid a full `terraform destroy` operation on your configuration.
- The `-replace` flag for `terraform plan` and `terraform apply` operations to safely recreate resources.

▼ **Move a resource to a different state file**

▼ The `terraform state mv` command allows to

▼ rename existing resources.

```
terraform state mv packet_device.worker packet_device.helper
```

▼ move a resource into a module.

```
terraform state mv packet_device.worker module.worker.packet_device.worker

// also change the resource name at the same time
terraform state mv packet_device.worker module.worker.packet_device.main
```

▼ move a module into a module.

```
terraform state mv module.app module.parent.module.app
```

- If renaming a resource or move it to another module & run terraform apply, terraform will apply & create the resource.
- state mv allows you to just change the reference so you can avoid a create & destroy action.

▼ Master the Workflow (TF commands)

▼ `terraform init`

▼ `terraform init` initializes your terraform projects by

- Downloading plugins dependencies eg. Providers & Modules.
- Creates a `.terraform` directory.
- Creates a dependency lock file to enforce expected versions for plugins & terraform itself.

- It is 1st command you will run for a new terraform project.
- If any modifications or changes to dependencies occurs, run `terraform init` again to apply the changes.
- `terraform init -upgrade` upgrades all plugins to the latest version that complies with the configuration's version constraint.
- `terraform init -get-plugins-false` skips plugin installation.
- `terraform init -plugin-dir=PATH` Force plugin installation to read plugins only from target directory.
- `terraform init -lockfile=MODE` Set a dependency lock file.
- `.terraform.lock.hcl` is a Dependency lock file.
- `.terraform.tfstate.lock.hcl` is a state lock file.

▼ `terraform get`

- The `terraform get` command downloads and updates modules in the root module.
- `terraform get` is lightweight because it only updates modules instead of initializing the state or pull new providers binaries.

▼ `terraform fmt`

- This command applies a subset of the terraform language style conventions along with other minor adjustments for readability.
- `terraform fmt` will by default look in to the current directory & apply formatting to all `.tf files`.
- `terraform fmt`
  - used for adjusting spacing two spaces indent.
  - will check & populate syntax errors.
  - `terraform fmt –diff` Display diffs of formatting changes & also supports other options.
  - `terraform fmt -recursive` is used to process the subdirectories as well.

▼ `terraform validate`

- The `terraform validate` runs checks that verify whether a configuration is syntactically valid & internally consistent, regardless of any provided variables or existing state.
- Validation requires an initialized working directory with any referenced plugins and modules installed.
- when you run `terraform plan` or `terraform apply`, validate will automatically be performed.

▼ `terraform console`

- `terraform console` is an `interactive shell` where you can evaluate expressions.

▼ `terraform plan`

- The `terraform plan` command creates an execution plan (aka Terraform Plan). Terraform plan consists of:
  - Reads the current state of any already-existing remote objects to make sure that the Terraform state is up-to-date.
  - Compares the current configuration to the prior state and noting any differences.
  - Proposes a set of change actions that should, if applied, make the remote objects match the configuration.
- Terraform's plan does not carry out the proposed changes.
- terraform plan file is a binary file that contains only machine code.

▼ Speculative Plans

- running `terraform apply`, terraform will output the description of the effect of the plan but without any intent to actually apply it.

▼ Saved Plans

- running `terraform apply -out=FILE`
- you can generate a saved plan file which you can then pass along to `terraform apply` eg. `terraform apply FILE` .

- when using saved plan, it will not prompt you to confirm & will act like auto-approve.

▼ `terraform apply`

- The `terraform apply` command executes the actions proposed in a `terraform plan`.

▼ Automatic Plan Mode

- when you run `terraform apply`.
- executes plan, validate & the apply.
- requires users to ***`manually approve`*** the plan by writing ***`yes`***.
- `terraform apply -auto-approve` flag will automatically approve the plan.

▼ Saved Plan Mode

- when providing filename with terraform apply i,e. `terraform apply FILE.`
- performs exactly the steps specified by that plan file.
- it does not prompt for approval, if you want to inspect a plan file before applying it, you can do it by using `terraform show`.

▼ `terraform destroy`

- The `terraform destroy` command is a convenient way to destroy all remote objects managed by a particular Terraform configuration.
- Can also create a ***`speculative destroy plan`***, to see what the effect of destroying would be.

▼ `terraform state <options>`

- The `terraform state` command is used for advanced state management. As 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.
- Usage: `terraform state <subcommand> [options] [args]`
- **Remote State:** The Terraform state subcommands all work with remote state just as if it was a local state. Reads and writes may take longer than normal as each read and each write do a full network roundtrip.
- **Backups**: All `terraform state` subcommands that modify the state write backup files. The path of these backup file can be controlled with `-backup`.

▼ commands

- `terraform state show 'resource name'`
- `terraform state list <resource name>`
- `terraform state list`

▼ `tererraform import [options] ADDRESS ID`

- The `terraform import` command importing existing resources into Terraform.
- Usage: `terraform import [options] ADDRESS ID`
- Import will find the existing resource from ID and import it into your Terraform state at the given ADDRESS.

- ID is dependent on the resource type being imported.
- Terraform will attempt to load configuration files that configure the provider being used for import.
- If no configuration files are present, Terraform will prompt you for access credentials & specify env variables to configure provider.

▼ **Import into Resource**

```
terraform import aws_instance.foo i-abcd1234
```

▼ **Import into Module**

```
terraform import module.foo.aws_instance.bar i-abcd1234
```

▼ **Import into Resource configured with count**

```
terraform import 'aws_instance.baz[0]' i-abcd1234
```

▼ **Import into Resource configured with for_each**

```
terraform import 'aws_instance.baz["example"]' i-abcd1234
```

▼ Saved Plan

- create saved plan by using `terraform plan my_saved_plan.plan`.
- apply the saved plan after review using `terraform apply my_saved_plan.plan`

▼ Configuration Drift

- Configuration Drift is when provisioned infrastructure has an unexpected configuration change dut to
  - team members manually adjusting configuration options.
  - malicious actors.
  - side effects from APIs, SDKs or CLIs.

> ⚠️ ***Configuration Drift*** going unnoticed could be loss or breach of cloud services & residing data or result in interruption of services or unexpected downtime.

- How to ***Detect*** Config drift?
  - A compliance tool that can detect mis-configuration eg. AWS Config, Azure Policies, GCP security Health Analytics.
  - Built-in support for drift detection eg. AWS CloudFormation Drift Detection.

    - Storing expected state eg. Terraform state files.

- How to `Correct` Config Drift?
    - A compliance tool that can remediate misconfiguration eg. AWS config.
    - Terraform refresh & plan commands.
    - Manually correcting the configuration (not recommended).
    - Tearing down & setting up the infrastructure again (risky).
- How to `Prevent` Config Drift?
    - Immutable Infrastructure, always creates & destroy, never re-use blue, green deployment strategy.
    - Using GitOps to version control IaC, & for new change/update making PR for review.

▼ Manage Resource Drift

- `Drift (Configuration or Infrastructure)` is when your expected resources are in a different state than your expected state.

▼ We can resolve Drift in 3 ways

▼ Replacing Resources

- When a resource has become damaged or degraded that cannot be detected by terraform we can use `-replace` flag.
- Why would want to mark a resource for replacement?
    - A cloud resource can become degraded or damaged & want to return to the expected resource to be healthy state.
- `terraform taint` is used to make a resource for replacement, but now it is deprecated.
- It is recommended & an alternative to use `-replace` flag & providing a resource address.
    - `terraform apply -replace="resouce_address"`
- A replace flag is available on plan & apply commands.
- The replace flag is used to work on only single resource.

▼ Importing Resources

- When an approved manual addition of resource needs to be added to our state file then we can use `terraform import` command.
- The `terraform import` command can be used to import existing resources into terraform.
- We need to
    - define a placeholder for the imported resource in the config file.
    - can leave the body blank & fill it after importing, it is not auto-filled.
    - `terraform import <resource-addr> <resouce-ID>`
    - `terraform import aws_instance.example i-elfnwef3`

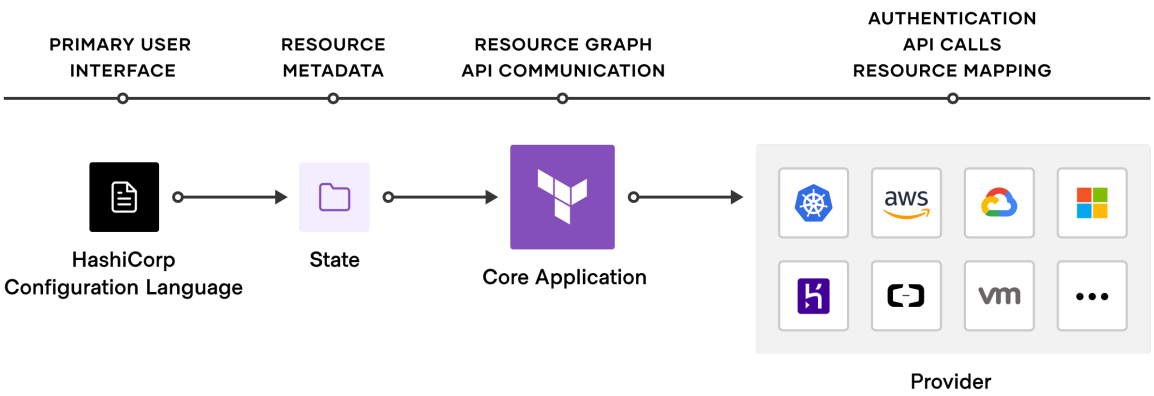
- this command only imports a resource one at a time.
- Not all resources are importable, must check docs before taking action.

▼ Refresh State

- When an approved manual configuration of a resource is changed or removed, we can use the `-refresh-only` flag to reflect changes in our state file.
- The terraform refresh command reads the current settings from all managed remote objects & updates the terraform state file to match.
- terraform refresh will not modify the remote infrastructure but modifies the terraform state file.
- `terraform refresh is deprecated` & with `-refresh-only` flag because this flag will be used to review the changes before taking action to update the state file which was missing in terraform refresh.
- the `-refresh-only` flag for terraform plan or apply allows you to refresh & update the state file without making changes to your remote infrastructure.

▼ Terraform Troubleshooting

- There are four potential types of issues that you could experience with Terraform



▼ They are 4 types of errors, Starting from the type of error closest to the user

▼ **Language Errors**

- When Terraform encounters a syntax error in the configuration for the `terraform or HCL language`, it prints out the line numbers and an explanation of the error.
- can be resolved using
  - `terraform fmt`.
  - `terraform validate`.
  - `terraform version`.

▼ **State errors**

- Your resources state has changed from the expected state in the configuration file.
- can be resolved using

        - terraform apply -refresh-only.
        - terraform -replace flag.
        - terraform import.

    ▼ **Core errors**

        - A bug has occurred with the core library.
        - Errors produced at this level may be a bug.
        - can be resolved using
            - TF_LOG logs.
            - & open GitHub issue.

    ▼ **Provider errors**

        - The provider's API has changed or does not work as expected due to emerging edge cases.
        - can be resolved using
            - TF_LOG logs.
            - & open GitHub issue.

    - Language errors & State errors are Easy to solve whereas Core & Provider errors are Harder to solve.

▼ **Terraform Debugging**

```
TF_LOG=TRACE TF_LOG_PATH=./terraform.log terraform apply
```

- Terraform has detailed logs which can be enabled by setting the TF_LOG environment variable to:
    - TRACE, DEBUG, INFO, WARN, ERROR, JSON.
    - outputs logs at the `TRACE` level or higher & used a parseable JSON encoding as the formatting.
- Logging can be enabled separately
    - `TF_LOG_CORE`
    - `TF_LOG_PROVIDER`
- takes the same option as TF_LOG
- `TF_LOG_PATH` to choose the logs file to debug & `store`.

▼ `crash.log`

- If terraform ever crashes, it saves a log file with the debug logs from the session as well as the panic message the backtrace to `crash.log`.
- This log file can be passed on to GitHub issue for raising & resolving.

▼ **Format the configuration**

```
terraform fmt
```

- The format command scans the current directory for configuration files and rewrites your Terraform configuration files to the recommended format.

▼ **Validate your configuration**

```
terraform validate
```

- `terraform fmt` only parses your HCL for interpolation errors or malformed resource definitions.
- `terraform validate` after formatting your configuration to check your configuration in the context of the provider's expectations.

▼ Cycle errors

- Cycle errors are instances of circular logic in the Terraform dependency tree.
- Terraform analyzes the dependencies between resources in your infrastructure configuration to determine the order to perform the operations.

▼ **Enable Terraform logging for Raising an Issue**

▼ **Bug reporting best practices**

- Experiencing errors due to provider or application issues. Once you eliminate the possibility of language misconfiguration, version mismatching, or state discrepancies, consider bringing your issue to the core Terraform team or Terraform provider community as a bug report.
- To provide the development team or the community working on your issue with full context for opening up a GitHub issue.

- `TRACE` provides the highest level of logging and contains all the information the development teams need. There are other logging levels, but are typically reserved for developers looking for specific information.

```
export TF_LOG_CORE=TRACE (or) Set-Item Env:TF_LOG_CORE TRAC
```

- You can also generate provider logs by setting the `TF_LOG_PROVIDER` environment variable. By including these in your bug reports, the provider development team can reproduce and debug provider specific errors.

```
export TF_LOG_PROVIDER=TRACE
```

- Once you have configured your logging, set the path for your error logs as an environment variable. If your `TF_LOG_CORE` or `TF_LOG_PROVIDER` environment variables are enabled, the `TF_LOG_PATH` variable will create the specified file and append logs generated by Terraform.

```
export TF_LOG_PATH=logs.txt
```

- To generate an example of the core and provider logs, run a `terraform refresh` operation.

```
terraform refresh
```

▼ Modules

- Modules in Terraform provide a way to organize and encapsulate related resources and enable the reuse of code and infrastructure configurations.
- Terraform modules are similar to the concepts of libraries, packages, or modules found in most programming languages, and provide many of the same benefits.
- In Terraform, a `module` is a collection of related resources and associated configuration that can be reused across different Terraform configurations, while a `resource` is a single infrastructure object, such as an EC2 instance, S3 bucket, or security group, that Terraform manages.
- Terraform will install any new modules in the `.terraform/modules` directory within your configuration's working directory.
- For local modules, Terraform will create a symlink to the module's directory. Because of this, any changes to local modules will be effective immediately, without having to reinitialize or re-run `terraform get`.
- They help to overcome several challenges associated with managing large and complex infrastructure:
  - ***Code Duplication***: Modules allow you to avoid duplicating code and configurations across multiple files or projects. By defining reusable modules.
  - ***Complexity***: Terraform modules allow you to abstract away the complexity of large and complex infrastructures, making it easier to manage and understand.
  - ***Collaboration***: Modules enable teams to work together more effectively, reduce errors & improve productivity by providing a common language and structure for managing infrastructure.
  - ***Scalability***: Modules allow you to easily scale your infrastructure as your needs grow. By defining reusable modules, you can quickly and easily spin up new environments and resources without having to write new code or configurations from scratch.

▼ What are modules for?

- Modules are used to organize and encapsulate code and configurations into reusable and independent units of functionality.
- They promote reuse, abstraction of complexity, and modularization, making it easier to manage and maintain code and infrastructure in a scalable and efficient way.

▼ What is a Terraform module?

- A Terraform module is a set of Terraform configuration files in a single directory.
- The simple configuration consisting of a single directory with one or more `.tf` files is a module & running tf commands in this directory is considered as `"root module"`.

- Terraform commands will only directly use the configuration files in one directory, which is usually the current working directory.
- Configuration can use module blocks to call modules in other directories, module that is called by another configuration is sometimes referred to as a `"child module"` of that configuration.

▼ **Local and remote modules**

- In Terraform, there are two types of modules: `local modules` and `remote modules`.
- `Local modules` are modules that exist on the local file system. They are typically defined in the same directory as the main Terraform configuration file, or in a subdirectory. Local modules are useful for organizing and encapsulating infrastructure code and configurations that are specific to a particular project or environment.
- `Remote modules` are modules that are stored in a remote location, such as a version control repository or a module registry. Remote modules can be referenced in the main Terraform configuration file using a URL, which Terraform will use to download the module from the remote location. Remote modules are useful for promoting code reuse and sharing infrastructure configurations across multiple projects and environments.
- Both local and remote modules provide the same basic functionality in Terraform, but remote modules are often preferred in larger and more complex infrastructures, where code reuse and sharing are more important. Additionally, remote modules can be versioned and updated independently of the main Terraform configuration, which can make it easier to manage and maintain infrastructure configurations over time.

▼ **Syntax**

```
module "consul" {
  source  = "hashicorp/consul/aws"
  version = "0.0.5"

  servers = 3
}
```

- This example configuration has two arguments:
  - source: it is required argument when using terraform module, in example module terraform will search for the module in given registry.
  - version: it is not a required argument but highly recommended to specify in the modules & it installs specified version modules for the project, If not specified latest modules will be installed & used.
  - Terraform treats other arguments in the module blocks as input variables for the module.
  - Modules can contain both required and optional arguments. You must specify all required arguments to use the module. Most module arguments correspond to the module's input variables.
  - Optional inputs will use the module's default values if not explicitly defined.

▼ **Modules Tutorials**

- **Use Registry Modules in Configuration**
- **Build and Use a Local Module**
- **Customize Modules with Object Attributes**
- ▼ **Share Modules in the Private Registry**
  - Adding/publishing created modules to the private registry in Terraform cloud.
  - Adding configuration code to the workspace, setting up Infrastructure, Provisioning the Infrastructure & deletion of it.
  - ▼ **Private Modules**

    📌 terraform-<PROVIDER>-<NAME>

    - In order to create & publish private modules, one must have VCS like github to store modules & terraform as a private registry (to import modules) to publish & access modules.
    - In order to publish modules to the Terraform registry, module names must have the format `terraform-<PROVIDER>-<NAME>`, where `<NAME>` can contain extra hyphens.
    - Terraform Cloud modules should be semantically versioned.

    📌 app.terraform.io/<ORGANIZATION-NAME>/terraform/<NAME>/<PROVIDER>

    - this above is the given format that modules from private registry can be referenced using registry source.
    - Even if providing variables in the Terraform cloud UI, creating & configuring required variables in variables.tf is recommended which makes easy collaboration & understanding the inputs.
    - Having outputs.tf is recommended to view/display the output variables in the terraform cloud.
- ▼ **Add Public Providers and Modules to your Private Registry**
  - Curating Public providers & Modules to consistently use for our growing infrastructure.
- **Refactor Monolithic Terraform Configuration**
- **Module Creation - Recommended Pattern**
- **Use Configuration to Move Resources**
- **Create and Use No-Code Modules**

▼ Additional Concepts InDepth

▼ Finding & Using Modules

- Public Registry Modules → <NAMESPACE>/<NAME>/<PROVIDER>
- Private Registry Modules → <HOSTNAME>/<NAMESPACE>/<NAME>/<PROVIDER>

▼ Module Versioning

- It is highly recommended to provide & use version constraints.
- Terraform's both public & private registry will have versions.
- Thirdparty providers registry may or may not have version constraint, If they have can use version in module block along with source else will be installed as is.

▼ Input 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.

```
// syntax
variable "input_variable_label_name" {
  type    = type of value
  default = actual_value
 }

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

- The name of a variable can be any valid identifier *except* the following: `source`, `version`, `providers`, `count`, `for_each`, `lifecycle`, `depends_on`, `locals` as these name are reserved for ***`meta arguments`***.
- The `type` argument in a `variable` block allows you to restrict the type of value that will be accepted as the value for a variable. If no type constraint is set then a value of any type is accepted.
- Type constraints are created from a mixture of type keywords and type constructors.
  - type keywords are
    - `string`
    - `number`
    - `bool`
  - type constructors allow you to specify complex types such as collections
    - `list(<TYPE>)`
    - `set(<TYPE>)`
    - `map(<TYPE>)`
    - `object({<ATTR NAME> = <TYPE>, ... })`
    - `tuple([<TYPE>, ...])`

- **Disallowing Null Input Values**
  - The `nullable` argument in a variable block controls whether the module caller may assign the value `null` to the variable.

```
variable "example" {
  type     = string
  nullable = false
}
```

-

▼ Output Variables

- Output values make information about your infrastructure available on the command line, and can expose information for other Terraform configurations to use. Output values are similar to return values in programming languages.
  - 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 an Output value

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

  - The `label` must be a Valid Identifier.
  - **Accessing Child Module Outputs**

```
module.<MODULE NAME>.<OUTPUT NAME>

module.web_server.instance_ip_addr
```

- **Custom Condition Checks**
  - Can use `precondition` blocks to specify guarantees about output data.
- `sensitive = true` this argument can be used to inadvertent exposure of sensitive information like db_username or passwords etc.
- `terraform output -json` can be used to parse the values into Human-Readable format. In this case, this format will expose the information of sensitive values.

▼ Calling a Child Module

- To *call* a module means to include the contents of that module into the configuration with specific values for its input variables. Modules are called from within other modules

using `module` blocks.

```
module "servers" {
  source = "./app-cluster"

  servers = 5
}
```

- **Accessing Module Output Values**

```
resource "aws_elb" "example" {
  # ...

  instances = module.servers.instance_ids
}
```

▼ The Core Terraform Workflow

▼ The Core Terraform workflow has three steps

1. **Write** - Author infrastructure as code.
2. **Plan** - Preview changes before applying.
3. **Apply** - Provision reproducible infrastructure.

- Workflow as a
  - ▼ Individual Practitioner (One Person Team)
    - ▼ Write
      - Writing TF configuration in editor of choice.
      - storing config code in VCS like GitHub.
      - Repeatedly running plan & validate for error-free syntaxes.
      - Tight FeedBack b/w editing code & running test commands.
    - ▼ Plan
      - When dev is confident with their work in the write step they commit their code to their local repo.
      - may be using only one main branch.
      - once commit is written they'll go to apply.
    - ▼ Apply
      - they will run terraform apply & review the plan.
      - after reviewing they'll approve the plan changes & await provisioning.
      - after successful provision, they will push the code to remote repo.
  - ▼ Teams using OSS
    - ▼ Write

- Each team members write their code locally on their machine in the editor of choice.
- A team member will store their code to a branch in their code repo.
  - Branches will avoid conflicts while a member is working on their code.
  - Branches will allow an opportunity to resolve conflict during merge into main.
- Terraform plan can be used as a quick fd back loop for small teams.
- For larger teams a concern over the sensitive credentials becomes a concern.
  - A CICD process can be implemented so burden of credentials can be abstracted away.

▼ Plan

- When a branch is ready to be incorporated on PR an execution plan can be generated & displayed within the PR for review.

▼ Apply

- To apply the changes the merge needs to be approved & merged, which will kick of a code build server that will run terraform apply.

- If the team starts from scratch, that too without terraform cloud.
- The DevOps team has to setup & maintain their own CI/CD pipeline.
- They have to figure out, how to store the state file (ex: standard remote backend).
- They are limited in their access controls (they can’t be granular).
- They have to figure out a way to safely store & inject secrets into their build server’s runtime env.
- Managing multiple Envs. can make the overhead of the infrastructure increase dramatically.

▼ Teams using Terraform Cloud

▼ Write

- A team will use tf cloud as their remote backend.
- Inputs variables will be stored on terraform cloud instead of the local machines.
- tf cloud integrates with VCS to quickly setup a CI/CD pipeline.
- A team member will write the code to branch & commits per usual.

▼ Plan

- A PR is created by team member & tf cloud will generate speculative plan for review in VCS. also can be reviewed & comment on the plan in tf cloud.

▼ Apply

- After PR is merged, tf cloud runtime env perform a terraform apply.
- A team member can confirm & apply changes.

- TF cloud streamlines a lot of CI/CD effort, storing & securing sensitive credentials & makes it easier to go back & audit multiple run history.

▼ Manage state

- Terraform uses the state to keep track of the infrastructure it manages. To use Terraform effectively, you have to keep your state accurate and secure.

▼ State management

▼ State Locking

- If supported by your backend, Terraform will lock your state for all operations that could write state. This prevents others from acquiring the lock and potentially corrupting your state.
- State locking happens automatically on all operations that could write state.
- Won't see any message that it is happening. If state locking fails, Terraform will not continue. You can disable state locking by using most of the commands with the `-lock` flag but it is not recommended.
- If acquiring the lock is taking longer than expected, Terraform will output a *`status message`*. If Terraform doesn't output a message, state locking is still occurring if your backend supports it.
- Not all backend supports state lock, check before configuring.

▼ Disabling Locking

- you can disable state locking for most commands with `-lock` flag but it is not recommended.

▼ Force Unlock

- Terraform has a force-unlock command to manually unlock the state if unlocking failed.
- If you unlock the state when someone else is holding the lock it could cause multiple writers. Force unlock should only be used to unlock your own lock in the situation where automatic unlocking failed.
- the `force-unlock` command requires a unique lock ID for locks & unlocks.
- `terraform force-unlock 12331e8o12e-dwkjnwe-1212 -force`

▼ Importance of state lock file?

- Terraform state lock file is used to prevent concurrent updates to the state by multiple Terraform processes or team members. When a Terraform command is run that modifies the state, it first acquires a state lock to ensure that no other Terraform process or team member is modifying the same state file at the same time. The state lock file is a marker file that is created in the same directory as the state file and contains information about the owner of the lock and the time the lock was acquired.
- In addition to preventing concurrent updates to the state, the state lock file can also be used to identify the owner of the lock and the time the lock was acquired. This can be useful for debugging purposes or when trying to troubleshoot issues related to state file updates.

▼ Sensitive Data in State

- Terraform state can contain sensitive data eg. long lived AWS credentials & is a possible attack vector for malicious actors.

- The state contains resource IDs and all resource attributes. For resources such as databases, this may contain initial passwords.
- local state,
  - When using local state, state is stored in plain-text JSON files.
  - need to be careful & this file must not be shared with anyone.
  - need to be careful & don't commit this file to VCS or git repository.
- remote state with terraform cloud,
  - When using terraform cloud remote backend, state is only ever held in memory & not persisted to disk.
  - the state file is encrypted at rest & encrypted in transit.
  - with terraform enterprise, you have detailed audit logging for tamper evidence.
- remote state with third-party storage,
  - you need to carefully review backed capabilities to determine if will meet your security & compliance requirements.

▼ **Recommendations**

- Manage any sensitive data with Terraform (like database passwords, user passwords, or private keys), treat the state itself as sensitive data.
- Storing state remotely can provide better security.
- Terraform does not persist state to the local disk when remote state is in use, and some backends can be configured to encrypt the state data at rest.

▼ For example

- Terraform Cloud always encrypts state at rest and protects it with TLS in transit. Terraform Cloud also knows the identity of the user requesting state and maintains a history of state changes. This can be used to control access and track activity.
- Terraform Enterprise also supports detailed audit logging.
- The S3 backend supports encryption at rest when the `encrypt` option is enabled. IAM policies and logging can be used to identify any invalid access. Requests for the state go over a TLS connection.

▼ **Terraform ignore file**

- When executing a remote plan or apply in a CLI driven run, an archive of your configuration directory is uploaded to terraform cloud.
- you can define the paths to ignore files from upload via `.terraformignore` file at the root of your configuration.
- if this file is not present, the archive will exclude the following by default,
  - .git.directories
  - .terraform/directories (exclusive of .terraform/modules)

> 💡 .terraformignore file just works like .gitignore

▼ **Command:** `-refresh-only`

- `terraform refresh` command will be deprecated & as alternative & recommended command to use is `terraform apply -refresh-only`
- we recommend using the following command in order to get the same effect but with the opportunity to review the changes that Terraform has detected before committing them to the state.
- This alternative command will present an interactive prompt for you to confirm the detected changes.

▼ **Backend management**

- Each terraform configuration can specify a backend, which defines where & how operations are performed, and where state snapshots are stored.

▼ Terraform Backends are divided into two type

`Standard backends`

- Only store state.
- Does not perform terraform operations eg. `terraform apply`.
- to perform operations, we need to use local CLI.
- third party backends are *`standard backends`* eg. `AWS S3`.

`Enhanced Backends`

- can both store state & can perform terraform operations.
- *`enhanced backends`* are sub-divided further.
  - `local` → files & data are stored on the local machine executing terraform commands.
  - `remote` → files & data are stored in the cloud eg. terraform cloud.

▼ Standard Backends

- Standard backends are third party providers other than terraform cloud.
- available backends are:
  - Cloud Services provides backends are —→ AWS S3 (locking via DynamoDB), Google Cloud Storage (GCS), AzureRM, Alibaba Cloud Object Services (OSS) (locking TableStore), Tencent CLoud Object Service (COS), Manta (Triton Object Service → TOS) etc.. these all support state locking.
  - Other than cloud services which provides backends are —→ HashiCrop consul (service n/w platform), etcd3, pg admin, kubernetes secrets these all provides lockingwhereas Artifactory(Universal Repo Manager) with no locking, HTTP protocol with optional locking etc..

▼ `local backend`

- The `local backend`
  - stores state in local filesystem.
  - locks the state using system APIs.
  - performs operations locally.
- by default, no specified backend

```
terraform {
}
```

- with backend

```
terraform {
  backend "local" {
    path = "./local path/specify here"
  }
}
```

- can set the backend to reference another state file so you can read its outputted values.

```
data "terraform_remote_state" "networking" {
  backend "local"
    config {
      path = "./local path/specify here"
  }
}
```

▼ `remote backend`

- A remote backend uses the terraform platform which is either:
  - Terraform Cloud
  - Terraform Enterprise
- with remote backend, when tf apply is performed via CLI the terraform cloud run environment is responsible for executing the operation.
- when using remote backend
  - all the environment vars are configured in tf cloud.
  - set a terraform cloud workspace.

  ▼ **Basic Configuration**

```
# Using a single workspace:
terraform {
  backend "remote" {
    hostname = "app.terraform.io"
    organization = "company"

    workspaces {
      name = "my-app-prod"
    }
```

```
  }
}

# Using multiple workspaces:
terraform {
  backend "remote" {
    hostname = "app.terraform.io"
    organization = "company"

    workspaces {
      prefix = "my-app-"
    }
  }
}
```

▼ Backend Initialization

- using CLI

```
# main.tf
terraform {
  required_version = "~> 0.12.0"

  backend "remote" {}
}
```

- using backend configuration file

```
# backend.hcl
workspaces { name = "workspace" }
hostname     = "app.terraform.io"
organization = "company"

// to access it
terraform init -backend-config=backend.hcl
```

▼ `terraform_remote_state`

- `terraform_remote_state` data source retrieves the root module output values from another terraform configuration, using latest snapshot from remote backend.

```
// remote backend

data "terraform_remote_state" "vpc" {
  backend = "remote"

    config = {
      organization = "hashicorp"
      workspaces = {
        name = "vpc-prod"
    }
  }
}

resource "aws_instance" "name" {
  subnet_id = data.terraform_remote_st
ate.vpc.outputs.subnet_id
}
```

```
// local backend

data "terraform_remote_state" "vpc"
{
  backend = "local"

    config = {
      path = "./statefile/pathhere"
  }
}

resource "aws_instance" "name" {
  subnet_id = data.terraform_remote
_state.vpc.outputs.subnet_id
}
```

- only the root-lvl output values from the remote state snapshots are exposed.
- resources data & output values from nested modules are not accessible.
- To make a nested module output value accessible as a root module output value, you must explicitly configure a passthrough in the root module.
- `terraform_remote_state` only exposes output values, it is recommend explicitly publish data for external consumption to a separate location instead of accessing it via remote state by this we will avoid making sensitive values as public.

▼ Built-in functions

▼ Numeric Functions

- abs
- floor
- log
- ceil
- min & max

▼ parseint

- parseint parses the given string as a representation of an integer in the specified base & returns the resulting number.

```
parseint(”100”, 10)
// output -> 100
```

▼ pow

- calculates an exponent, by raising its first argument to the power of the second argument.

```
pow(3, 2)
// output -> 9

pow(4, 0)
// output -> 1
```

▼ signum

- signum determines the sign of a number, returning a number between -1 and 1 to represent the sign.

```
signum(-13)
// -1

signum(0)
// 0

signum(334)
//1
```

▼ String Functions

▼ chomp

- chomp removes newline characters at the end of a string.

```
> chomp("hello\n")
hello

> chomp("hello\r\n")
hello

> chomp("hello\n\n")
hello
```

▼ format

- format produces a string by formatting a number of other values according to a specification string.

```
 > format("Hello, %s!", "Ander")
Hello, Ander!

> format("There are %d lights", 4)
There are 4 lights
```

▼ formatlist

- formatlist produces a list of strings by formatting a number of other values according to a specification string.

```
 // syntax
formatlist(spec, values...)
```

```
> formatlist("Hello, %s!", ["Valentina", "Ander", "Olivia", "Sam"])
[
  "Hello, Valentina!",
  "Hello, Ander!",
  "Hello, Olivia!",
  "Hello, Sam!",
]
> formatlist("%s, %s!", "Salutations", ["Valentina", "Ander", "Olivia", "Sam"])
[
  "Salutations, Valentina!",
  "Salutations, Ander!",
  "Salutations, Olivia!",
  "Salutations, Sam!",
]
```

▼ indent

- indent adds a given number of spaces to the beginnings of all the first line in a given multi-line string.

```
// syntax
indent(num_spaces, string)

// examples
> "  items: ${indent(2, "[\n  foo,\n  bar,\n]\n")}"
  items: [
    foo,
    bar,
  ]
```

▼ join

- join produces a string by concatenating together all elements of a given list of strings with the given delimiter.

```
// syntx
join(separator, list)


// examples
> join(", ", ["foo", "bar", "baz"])
foo, bar, baz


> join(", ", ["foo"])
foo
```

▼ lower

- `lower` converts all cased letters in the given string to lowercase.

```
> lower("HELLO")
hello

> lower("АЛЛО!")
алло!
```

▼ regex

- `regex` applies a regular expression to a string and returns the matching substrings.

```
// syntax
regex(pattern, string)

// examples
> regex("[a-z]+", "53453453.345345aaabbbccc23454")
aaabbbccc


> regex("(\\d\\d\\d\\d)-(\\d\\d)-(\\d\\d)", "2019-02-01")
[
  "2019",
  "02",
  "01",
]

> regex("^(?:(?P<scheme>[^:/?#]+):)?(?://(?P<authority>[^/?#]*))?", "https://terraform.io/do
cs/")
{
```

```
  "authority" = "terraform.io"
  "scheme" = "https"
}

> regex("[a-z]+", "53453453.34534523454")

Error: Error in function call

Call to function "regex" failed: pattern did not match any part of the given
string.
```

▼ regexall

- `regexall` applies a regular expression to a string and returns a list of all matches.

```
// syntax
regexall(pattern, string)

// examples
> regexall("[a-z]+", "1234abcd5678efgh9")
[
  "abcd",
  "efgh",
]

> length(regexall("[a-z]+", "1234abcd5678efgh9"))
2

> length(regexall("[a-z]+", "123456789")) > 0
false
```

▼ replace

- `replace` searches a given string for another given substring, and replaces each occurrence with a given replacement string.

```
// syntax
replace(string, substring, replacement)

// examples
> replace("1 + 2 + 3", "+", "-")
1 - 2 - 3

> replace("hello world", "/w.*d/", "everybody")
hello everybody
```

▼ split

- `split` produces a list by dividing a given string at all occurrences of a given separator.

```
// syntax
split(separator, string)

// examples
> split(",", "foo,bar,baz")
[
  "foo",
  "bar",
  "baz",
]
```

```
> split(",", "foo")
[
  "foo",
]


> split(",", "")
[
  "",
]
```

▼ strrev

- `strrev` reverses the characters in a string.

```
// syntax
strrev(string)

// examples
> strrev("hello")
olleh

> strrev("a ☃")
☃ a
```

▼ substr

- `substr` extracts a substring from a given string by offset and (maximum) length.

```
// syntax
substr(string, offset, length)

// examples
> substr("hello world", 1, 4)
ello

> substr("🤔🤷", 0, 1)
🤔

> substr("hello world", -5, -1)
world

> substr("hello world", 6, 10)
world
```

▼ title

- `title` converts the first letter of each word in the given string to uppercase.

```
> title("hello world")
Hello World
```

▼ trim

- `trim` removes the specified set of characters from the start and end of the given string.

```
// syntax
trim(string, str_character_set)

// examples
> trim("?!hello?!", "!?")
"hello"

> trim("foobar", "far")
"oob"

> trim("   hello! world.!  ", "! ")
"hello! world."
```

▼ trimprefix

- `trimprefix` removes the specified prefix from the start of the given string. If the string does not start with the prefix, the string is returned unchanged.

```
> trimprefix("helloworld", "hello")
world

> trimprefix("helloworld", "cat")
helloworld
```

▼ trimsuffix

- `trimsuffix` removes the specified suffix from the end of the given string.

```
> trimsuffix("helloworld", "world")
hello
```

▼ trimspace

- `trimspace` removes any space characters from the start and end of the given string.

```
> trimspace("  hello\n\n")
hello
```

▼ Collection Functions

▼ alltrue

- `alltrue` returns `true` if all elements in a given collection are `true` or `"true"`. It also returns `true` if the collection is empty.

```
// syntax
alltrue(list)

// examples
> alltrue(["true", true])
true
> alltrue([true, false])
false
```

▼ anytrue

- `anytrue` returns `true` if any element in a given collection is `true` or `"true"`. It also returns `false` if the collection is empty.

```
// syntax
anytrue(list)

// examples
> anytrue(["true"])
true
> anytrue([true])
true
> anytrue([true, false])
true
> anytrue([])
false
```

▼ chunklist

- `chunklist` splits a single list into fixed-size chunks, returning a list of lists.

```
// syntax
chunklist(list, chunk_size)

// examples
> chunklist(["a", "b", "c", "d", "e"], 2)
[
  [
    "a",
    "b",
  ],
  [
    "c",
    "d",
  ],
  [
    "e",
  ],
]


> chunklist(["a", "b", "c", "d", "e"], 1)
[
  [
    "a",
  ],
  [
    "b",
  ],
  [
    "c",
  ],
  [
    "d",
  ],
  [
    "e",
  ],
]
```

▼ coalesce

- `coalesce` takes any number of arguments and returns the first one that isn't null or an empty string.

```
> coalesce("a", "b")
a
> coalesce("", "b")
b
> coalesce(1,2)
1
```

▼ coalescelist

- `coalescelist` takes any number of list arguments and returns the first one that isn't empty.

```
> coalescelist(["a", "b"], ["c", "d"])
[
  "a",
  "b",
]


> coalescelist([], ["c", "d"])
[
  "c",
  "d",
]
```

▼ compact

- `compact` takes a list of strings and returns a new list with any empty string elements removed.

```
> compact(["a", "", "b", "c"])
[
  "a",
  "b",
  "c",
]
```

▼ concat

- `concat` takes two or more lists and combines them into a single list.

```
> concat(["a", ""], ["b", "c"])
[
  "a",
  "",
  "b",
  "c",
]
```

▼ contains

- `contains` determines whether a given list or set contains a given single value as one of its elements.

```
// syntax
contains(list, value)

// examples
> contains(["a", "b", "c"], "a")
true

> contains(["a", "b", "c"], "d")
false
```

▼ distinct

- `distinct` takes a list and returns a new list with any duplicate elements removed.

```
> distinct(["a", "b", "a", "c", "d", "b"])
[
  "a",
  "b",
  "c",
  "d",
]
```

▼ element

- `element` retrieves a single element from a list.

```
// syntax
element(list, index)

// examples
> element(["a", "b", "c", "d"], 2)
c

// greater then given list, hence gives zero-based index value.
> element(["a", "b", "c", "d"], 4)
a

// greater then given list, hence gives zero-based index value.
> element(["a", "b", "c"], 3)
a

> element(["a", "b", "c"], -1)
error: cannot use negative index

> element(["a", "b", "c"], length(["a", "b", "c"])-1)
c
```

▼ index

- `index` finds the element index for a given value in a list.

```
// syntax
index(list, value)

// examples
> index(["a", "b", "c"], "b")
1
```

▼ flatten

- `flatten` takes a list and replaces any elements that are lists with a flattened sequence of the list contents.

```
> flatten([["a", "b"], [], ["c"]])
["a", "b", "c"]

> flatten([[["a", "b"], []], ["c"]])
["a", "b", "c"]
```

▼ keys

- `keys` takes a map and returns a list containing the keys from that map.

```
> keys({a=1, c=2, d=3})
[
  "a",
  "c",
  "d",
]
```

▼ length

- `length` determines the length of a given list, map, or string.

```
> length([])
0

> length(["a", "b"])
2

> length({"a" = "b"})
1

> length("hello")
5

> length("👾🕹")
2
```

▼ lookup

- `lookup` retrieves the value of a single element from a map, given its key. If the given key does not exist, the given default value is returned instead.

```
// syntax
lookup(map, key, default)

// examples
> lookup({a="ay", b="bee"}, "a", "what?")
ay

> lookup({a="ay", b="bee"}, "c", "what?")
what?
```

▼ matchkeys

- `matchkeys` constructs a new list by taking a subset of elements from one list whose indexes match the corresponding indexes of values in another list.

```
// syntax
matchkeys(valueslist, keyslist, searchset)

// examples
> matchkeys(["i-123", "i-abc", "i-def"], ["us-west", "us-east", "us-east"], ["us-east"])
[
  "i-abc",
  "i-def",
]
```

▼ merge

- `merge` takes an arbitrary number of maps or objects, and returns a single map or object that contains a merged set of elements from all arguments.

```
> merge({a="b", c="d"}, {e="f", c="z"})
{
  "a" = "b"
  "c" = "z"
  "e" = "f"
}


> merge({a="b"}, {a=[1,2], c="z"}, {d=3})
{
  "a" = [
    1,
    2,
  ]
  "c" = "z"
  "d" = 3
}
```

▼ one

- `one` takes a list, set, or tuple value with either zero or one elements. If the collection is empty, `one` returns `null`. Otherwise, `one` returns the first element. If there are two or more elements then `one` will return an error.

```
> one([])
null

> one(["hello"])
"hello"

> one(["hello", "goodbye"])

Error: Invalid function argument

Invalid value for "list" parameter: must be a list, set, or tuple value with
either zero or one elements.
```

▼ range

- `range` generates a list of numbers using a start value, a limit value, and a step value.

```
// syntax
range(max)
range(start, limit)
range(start, limit, step)

// examples
> range(3)
[
  0,
  1,
  2,
]

> range(1, 4)
[
  1,
  2,
  3,
]

> range(1, 8, 2)
[
  1,
  3,
  5,
  7,
]

> range(1, 4, 0.5)
[
  1,
  1.5,
  2,
  2.5,
  3,
  3.5,
]

> range(4, 1)
[
  4,
  3,
  2,
]

> range(10, 5, -2)
[
  10,
  8,
  6,
]
```

▼ reverse

- `reverse` takes a sequence and produces a new sequence of the same length with all of the same elements as the given sequence but in reverse order.
- similar to `strrev` in string functions.

```
> reverse([1, 2, 3])
[
  3,
```

```
  2,
  1,
]
```

▼ setintersection

- it computes the intersection of the sets.

```
// syntax
setintersection(sets...)

// examples
> setintersection(["a", "b"], ["b", "c"], ["b", "d"])
[
  "b",
]
```

▼ setproduct

- The `setproduct` function finds all of the possible combinations of elements from all of the given sets by computing the Cartesian product.

```
// syntax
setproduct(sets...)

// examples
> setproduct(["development", "staging", "production"], ["app1", "app2"])
[
  [
    "development",
    "app1",
  ],
  [
    "development",
    "app2",
  ],
  [
    "staging",
    "app1",
  ],
  [
    "staging",
    "app2",
  ],
  [
    "production",
    "app1",
  ],
  [
    "production",
    "app2",
  ],
]
```

▼ setsubtract

- The `setsubtract` function returns a new set containing the elements from the first set that are not present in the second set. In other words, it computes the relative complement of the second set.

```
// syntax
setsubtract(a, b)

// examples
> setsubtract(["a", "b", "c"], ["a", "c"])
[
  "b",
]
```

▼ setunion

- it computes the union of the sets.

```
//syntax
setunion(sets...)

// examples
> setunion(["a", "b"], ["b", "c"], ["d"])
[
  "d",
  "b",
  "c",
  "a",
]
```

▼ slice

- `slice` extracts some consecutive elements from within a list.

```
// syntax
slice(list, startindex, endindex)

// examples
> slice(["a", "b", "c", "d"], 1, 3)
[
  "b",
  "c",
]
```

▼ values

- `values` takes a map and returns a list containing the values of the elements in that map.

```
> values({a=3, c=2, d=1})
[
  3,
  2,
  1,
]
```

▼ transpose

- `transpose` takes a map of lists of strings and swaps the keys and values to produce a new map of lists of strings.

```
> transpose({"a" = ["1", "2"], "b" = ["2", "3"]})
{
  "1" = [
    "a",
  ],
  "2" = [
    "a",
    "b",
  ],
  "3" = [
    "b",
  ],
}
```

▼ zipmap

- `zipmap` constructs a map from a list of keys and a corresponding list of values.

```
// syntax
zipmap(keyslist, valueslist)

// examples
> zipmap(["a", "b"], [1, 2])
{
  "a" = 1
  "b" = 2
}
```

▼ Encoding Functions

- Functions that will `encode` & `decode` for various formats.
- https://developer.hashicorp.com/terraform/language/functions/base64decode

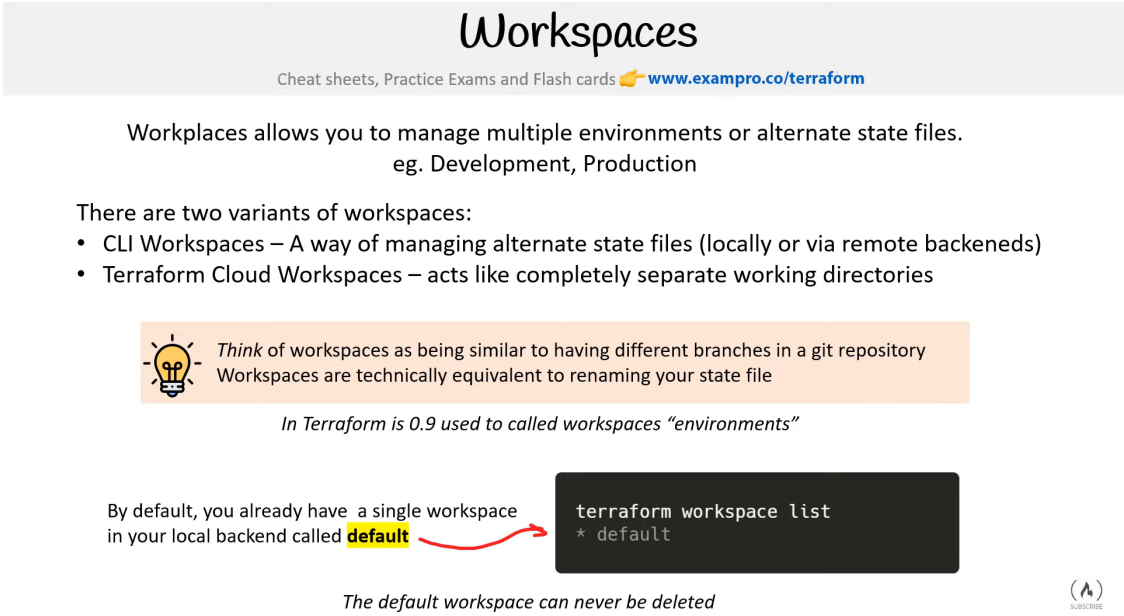
▼ Filesystem Functions

- https://developer.hashicorp.com/terraform/language/functions

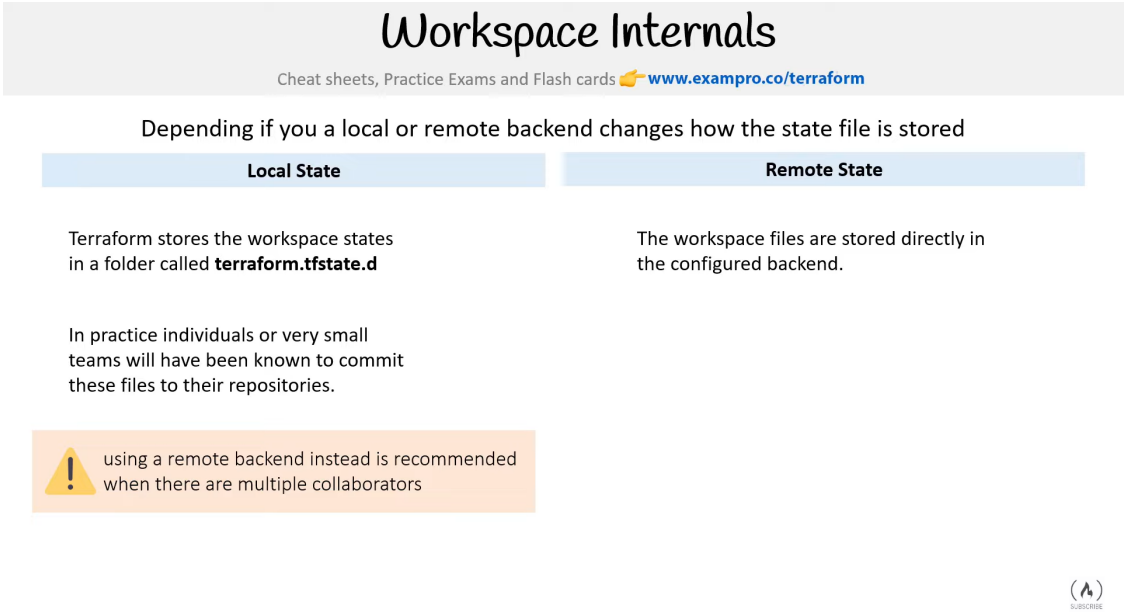
- Date & Time Functions
- Hash & Crypto Functions
- IP Network Functions

▼ Workspace in Terraform Cloud

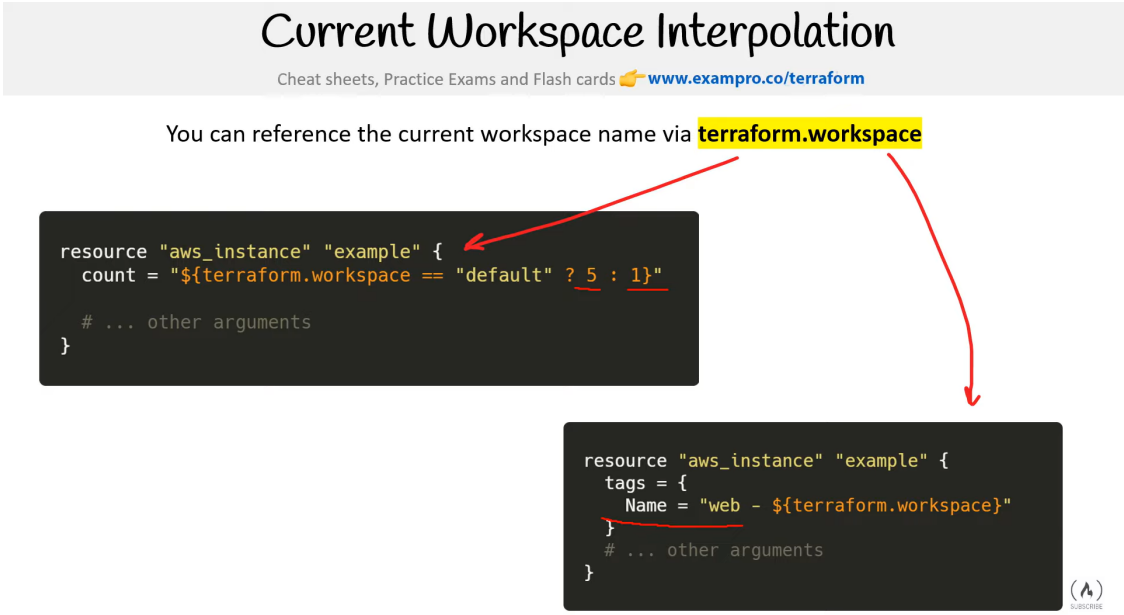
▼ WorkSpaces



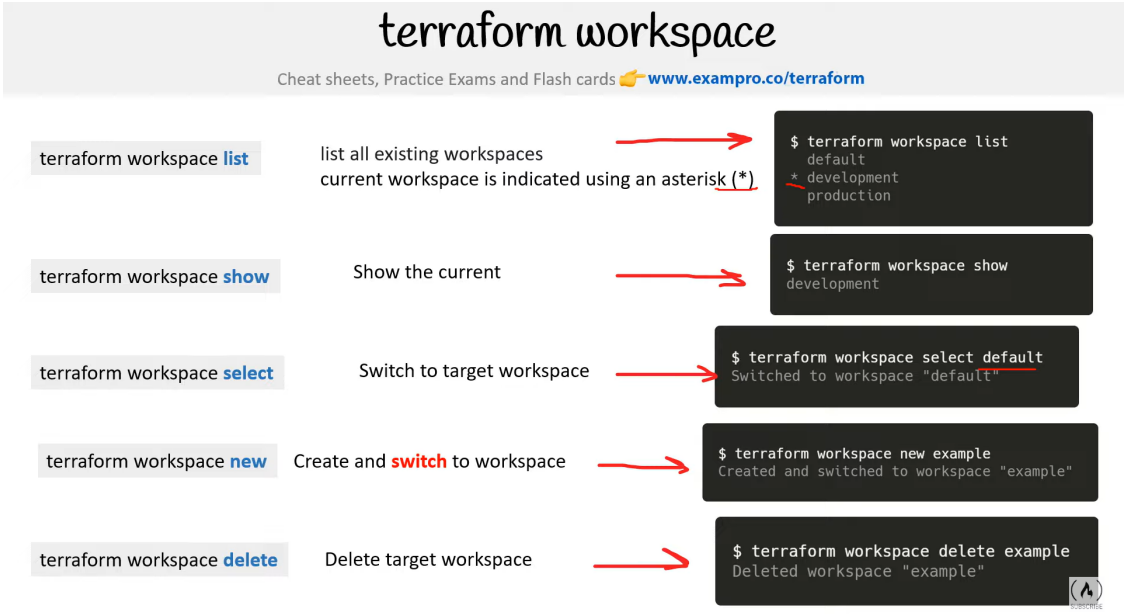
▼ Workspaces Internals



▼ Current Workspace Interpolation



▼ Terraform Workspace CLI Commands



▼ Multiple Workspaces

Multiple Workspaces

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

A Terraform configuration has a backend that:

- defines **how operations are executed**
- where **persistent data is stored** e.g. Terraform State

Multiple workspaces are currently supported by the following backends

- AzureRM
- Consul
- COS
- GCS
- Kubernetes
- Local
- Manta
- Postgres
- Remote
- S3

Certain backends support *multiple* named workspaces

- allowing multiple states to be associated with a single configuration.

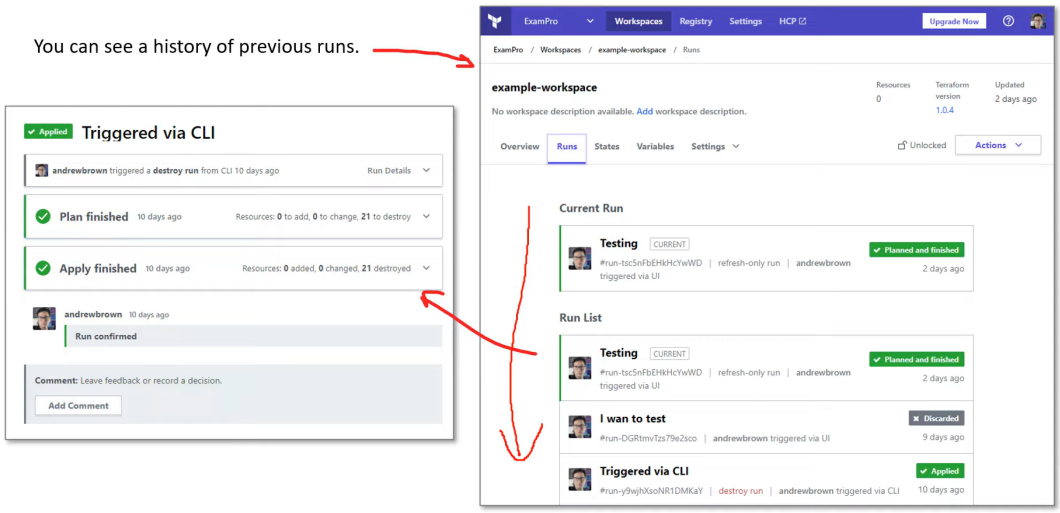
The configuration still has only one backend, but multiple distinct instances of that configuration to be deployed without configuring a new backend or changing authentication credentials.

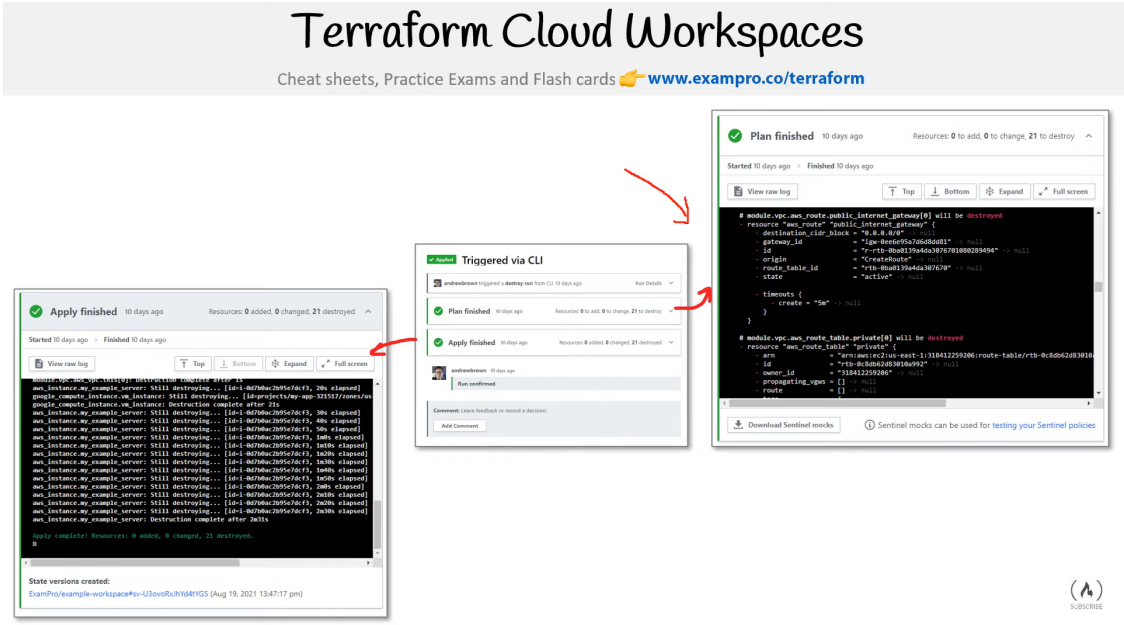
▼ Terraform Cloud Workspaces

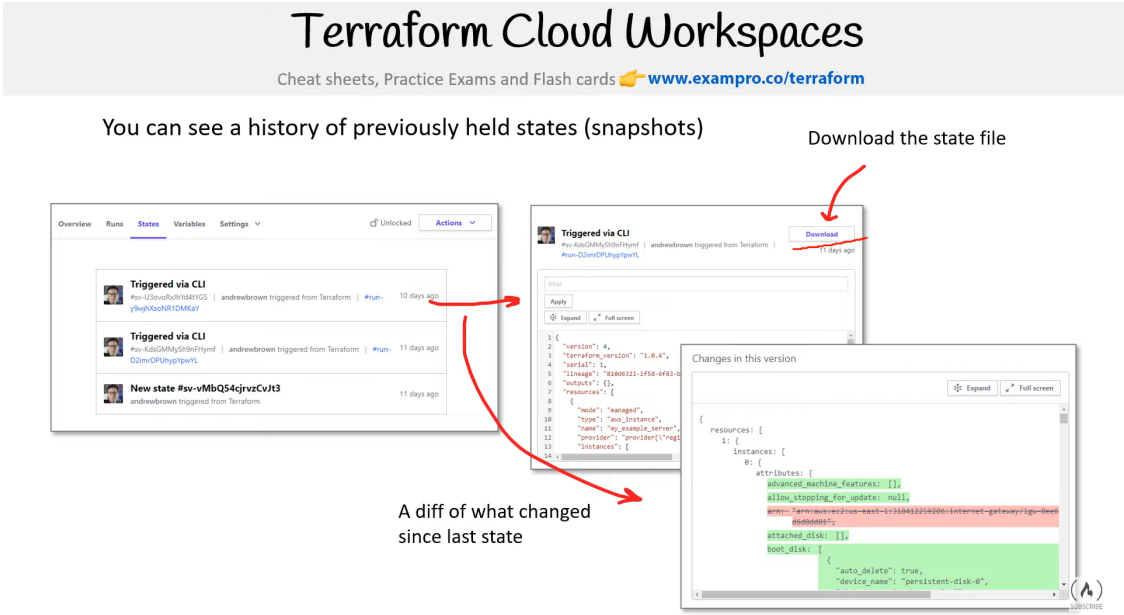
Terraform Cloud Workspaces

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

You can see a history of previous runs.







▼ Terraform Cloud Run Triggers

Terraform Cloud Run Triggers

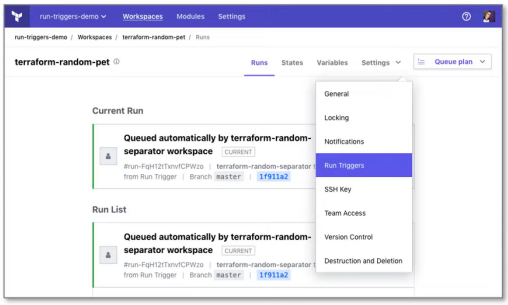
Cheat sheets, Practice Exams and Flash cards 👉www.exampro.co/terraform

Terraform Cloud provides a way to **connect your workspace to one or more workspaces** via **Run Triggers** within your organization, known as "source workspaces".

Run Triggers
allow runs to queue automatically in your workspace on successful apply of runs in any of the source workspaces. You can connect each workspace to up to 20 source workspaces.

Run triggers are designed for workspaces that rely on information or infrastructure produced by other workspaces.

If a Terraform configuration uses data sources to read values that might be changed by another workspace, run triggers let you explicitly specify that external dependency.



▼ Workspace Differences

Workspaces Differences

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

Terraform Cloud workspaces and local working directories serve the same purpose, but they store their data differently:

Component	Local Terraform	Terraform Cloud
Terraform configuration	On disk	In linked version control repository, or periodically uploaded via API/CLI
Variable values	As .tfvars files, as CLI arguments, or in shell environment	In workspace
State	On disk or in remote backend	In workspace
Credentials and secrets	In shell environment or entered at prompts	In workspace, stored as sensitive variables

▼ Terraform Cloud

▼ Org level Permissions

Terraform Cloud - Organization-Level Permissions

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

Organization-Level Permissions manage certain resources or settings **across an organization**

- **Manage Policies** - create, edit, and delete the organization's Sentinel policies
- **Manage Policy Overrides** - override soft-mandatory policy checks.
- **Manage Workspaces** - create and administrate all workspaces within the organization
- **Manage VCS Settings** - set of VCS providers and SSH keys available within the organization

Organization Owners
Every organization has organization owner(s).
This is a special role that has every available permission and some actions only available to owners:

- Publish private modules
- Invite users to organization
- Manage team membership
- View all secret teams
- Manage organization permissions
- Manage all organization settings
- Manage organization billing
- Delete organization
- Manage agent

▼ WorkSpace Level Permissions

Terraform Cloud - Workspace-Level Permissions

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

Workspace-Level Permissions manage resource and settings for a **specific workspace**

General Workspace Permissions
Granular permissions you can apply to a user via **custom workspace permissions**

- Read runs
- Queue plans
- Apply runs
- Lock and unlock workspaces
- Download sentinel mocks
- Read variable
- Read and write variables
- Read state outputs
- Read state versions
- Read and write state versions

Fixed Permission Sets
Premade permissions for quick assignment.

- **Read**
  - Read runs
  - Read variables
  - Read state versions
- **Plan**
  - Queue Plans
  - Read variables
  - Read state versions
- **Write**
  - Apply runs
  - Lock and unlock workspaces
  - Download Sentinel mocks
  - Read and write variables
  - Read and write state versions

Workspace Admins
A workspace admin is a special role that grants the all level of permissions and some workspace-admin-only permissions:

- Read and write workspace settings
- Set or remove workspace permissions of any team
- Delete workspace

▼ API Tokens

Terraform Cloud – API Tokens

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

Terraform Cloud supports three types of API Tokens. User, Team and Organization Tokens.

Organization API Tokens
- Have permissions across the entire organization
- Each organization can have one valid API token at a time
- Only organization owners can generate or revoke an organization's token.
- Organization API tokens are designed for creating and configuring workspaces and teams.
  - Not recommended as an all-purpose interface to Terraform Cloud

Team API Tokens
- allow access to the workspaces that the team has access to, without being tied to any specific user
- Each team can have **one** valid API token at a time
- any member of a team can generate or revoke that team's token
- When a token is regenerated, the previous token immediately becomes invalid
- designed for performing API operations on workspaces.
- same access level to the workspaces the team has access to

User API Tokens
- Most flexible token type because they inherit permissions from the user they are associated with
- Could be for a real user or a machine user.

▼ Private Registry

Terraform Cloud – Private Registry

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

Terraform Cloud allows you to **publish private modules** for your Organization within the **Terraform Cloud Private Registry**

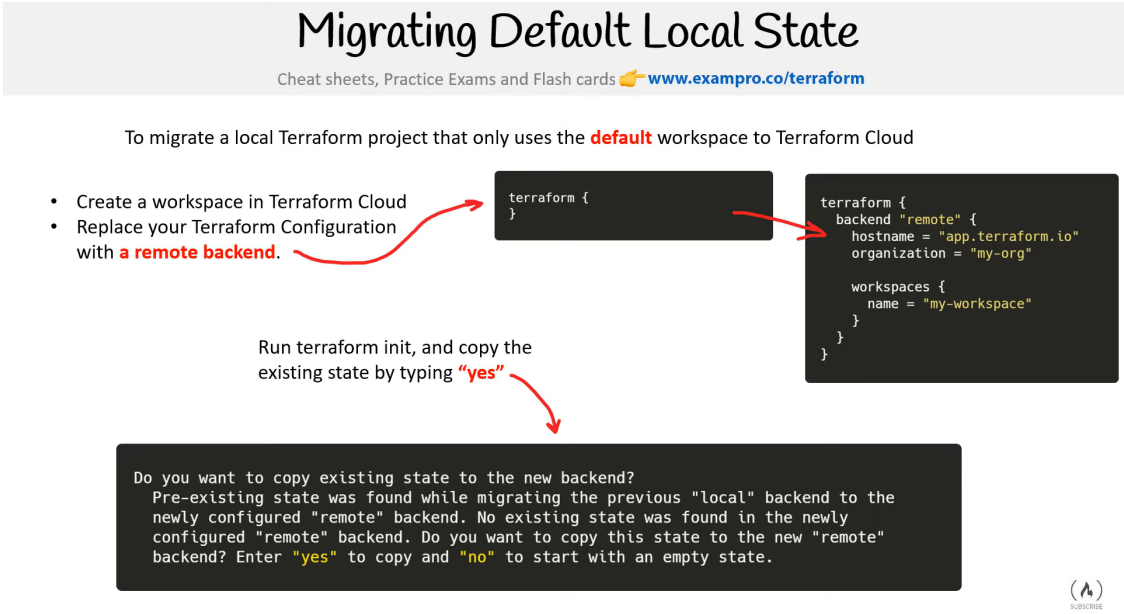
Terraform Cloud's private module registry helps you share Terraform modules across your organization.
It includes support for:
- module versioning
- a searchable and filterable list of available modules
- a configuration designer

All users in your organization can view your private module registry.

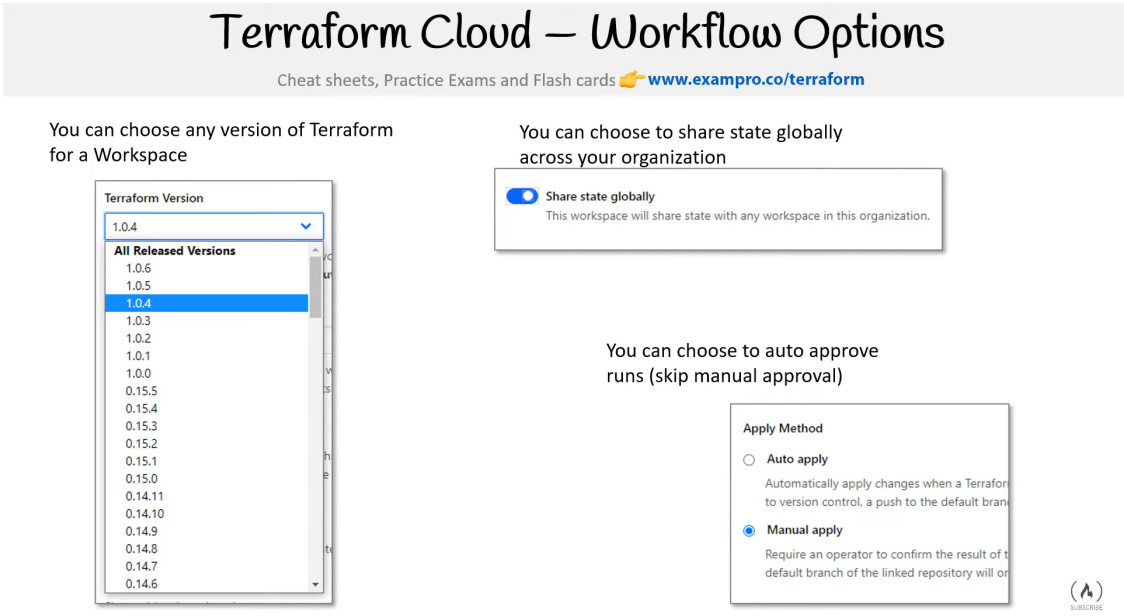
Authentication
You can use either a user token or a team token for authentication, but the type of token you choose may grant different permissions.

Using terraform login will obtain a user token
To use a team token you'll need to manually set it your terraform CLI configuration file

▼ Migrating Default Local State



▼ Terraform Cloud - WorkFlow Options



▼ Terraform Cloud Run Environment

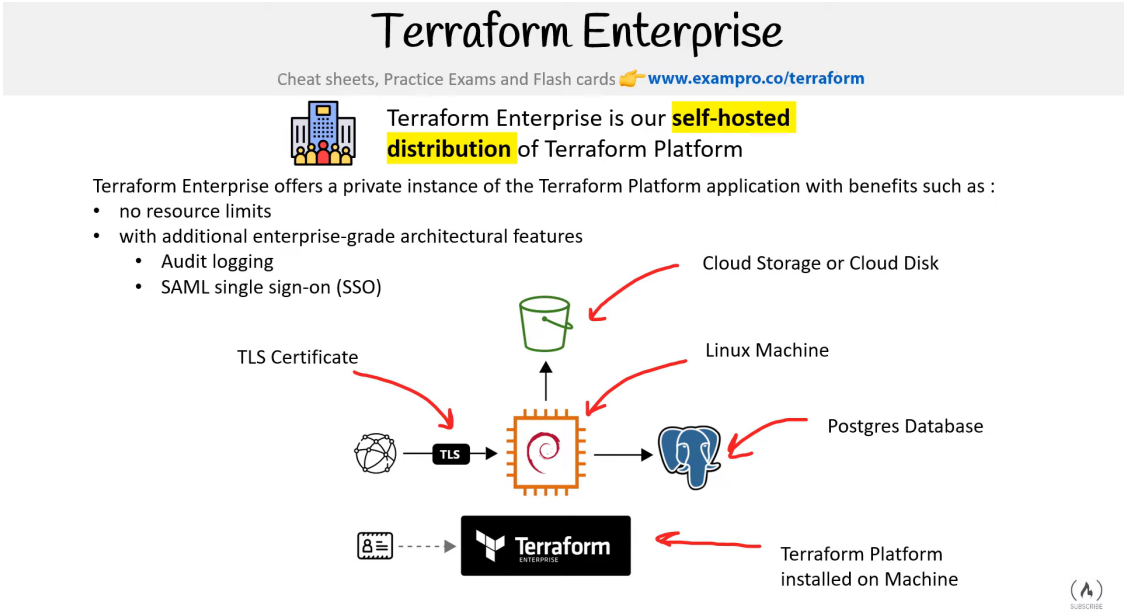
▼ Terraform Cloud Agents

▼ Sentinel CLI

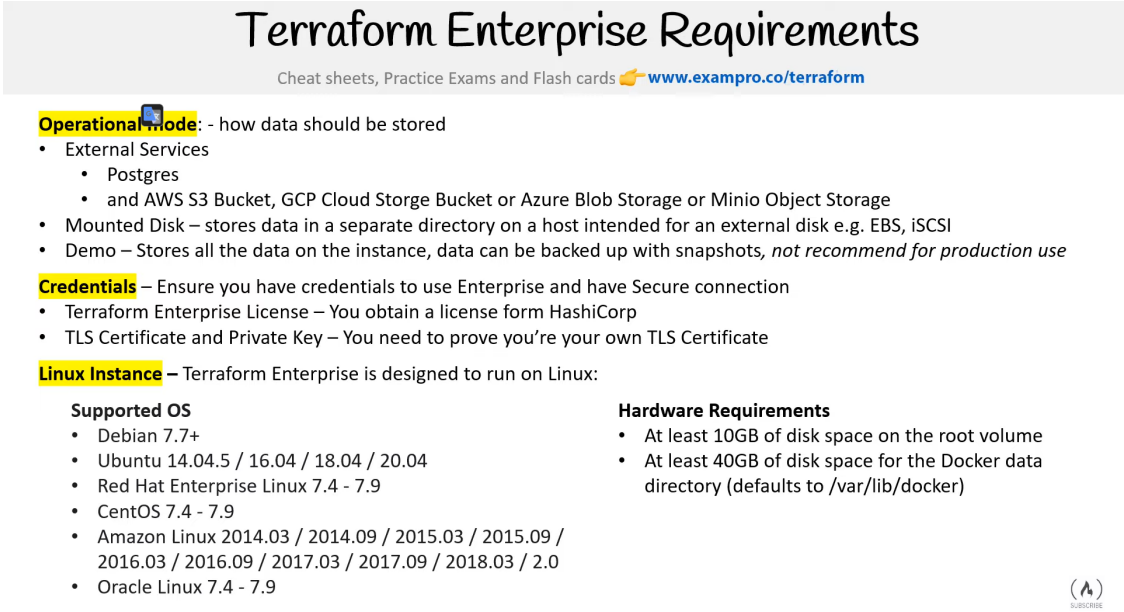
- Terraform Cloud uses Sentinel as part of Teams & Governance to enable granular policy control for your infrastructure.
- Sentinel is a language and policy framework, which restricts Terraform actions to defined, allowed behaviors.
- Sentinel Policy is available for Hashicorp Terraform Enterprise customers only.
- `Policy authors` manage Sentinel policies in Terraform Cloud with policy sets, which are groups of policies. `Organization owners` control the scope of policy sets by applying certain policy sets to the entire organization or to select workspaces.

- The `Sentinel CLI` validates and tests rules.

▼ Terraform Enterprise



▼ Terraform Enterprise Requirements



▼ AIR gapped Envs

Air Gapped Environments

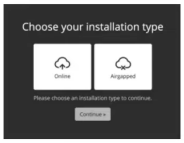
Cheat sheets, Practice Exams and Flash cards 👉 www.exampro.co/terraform

What is Air Gap?

Air Gap or disconnected network is a network security measure employed on one or more computers to ensure that a secure computer network is physically isolated from unsecure networks e.g. Public Internet

No internet. No outside connectivity

Industries in the **Public Sector** (e.g. government, military) or **large enterprise** e.g. Finance and Energy often employ air gap networks.



HashiCorp Terraform Enterprise **supports** an installation type for Air Gapped Environments

To install or update Terraform Enterprise you will supply an "air gap bundle" which is an archive of a Terraform Enterprise release version

▼ Terraform Cloud Features & Planning

Terraform Cloud Features and Pricing

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

Open-Source Software (OSS)	Cloud			Self-Hosted
	Free	Teams and Governance	Business	Enterprise (self-hosted)
IaC, Workspaces, Variables, Runs, Resource Graph, Providers, Modules, Public Module Registry				
	Remote State, VCS Connection, Workspace Mgmt., Secure Variable Storage, Remote Runs, Private Module Registry			
		Team Management, **Sentinel Policy as Code Management**, Cost Estimation		
			Single Sign On (SSO), Audit Logging	
			Self Hosted Agents	
			Configuration Designer, ServiceNow Integration	
	1 Current Runs	2 Current Runs	Unlimited Current Runs	
Local CLI	Cloud			Private
Community		Bronze	Silver, Gold	
	$0 up to 5 users	Starting at $20 user/month	Contact Sales	Contact Sales

▼ Terraform & Consul

Terraform and Consul

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

Consul is a **service networking platform** which provides:

- **service discovery** – central registry for services in the network
  - Allows for direct communication, no single-point of failure via load balancers
- **service mesh** – managing network traffic between services
  - A communication layer on top of your container application, *think middleware*
- **application configuration capabilities**

Consul is useful when you have a micro-service or **service-oriented architecture** with **hundred or thousands of services** (containerized apps or workloads)

Consul integrates with Terraform in the following ways:

- Remote backend
  - Consul has a Key Value (KV) Store to store configurations
- Consul Provider

▼ Terraform & Vault

HashiCorp Vault

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

What is HashiCorp Vault?

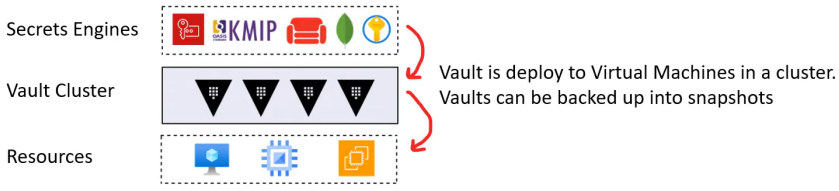
Vault is a tool for **securely accessing secrets** from multiple secrets data stores.

Vault is deployed to a server where:

- Vault Admins can directly manage secrets
- Operators (developers) can access secrets via an API

Vault provides a unified interface:

- to any secret
  - AWS Secrets, Consul Key Value, Google Cloud KMS, Azure Service principles....
- providing tight access control
  - Just-in-Time (JIT) - reducing surface attack based on range of time
  - Just Enough Privilege (JeP) - reducing service attack by providing least-permissive permissions
- recording a detailed audit log – tamper evidence



Terraform and Vault

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

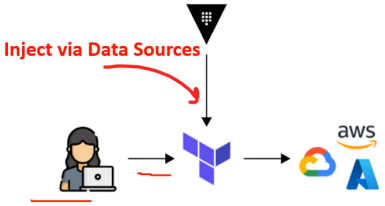
When a developer is working with Terraform and they need to deploy to a provider eg. AWS, they will need AWS credentials

AWS Credentials are long-lived, meaning a user generates a key and secret, and they are usable until they are deleted.

The AWS Credentials reside on the developer's local machine, and so that machine is at risk of being compromised by a malicious actors looking to steal the credentials.

If we could:
- provide credentials Just-In-Time (JIT)
- expire the credentials after a short amount of time (short-lived)

we can reduce the attack surface area of the local machine.



Vault can be used to **inject** *short-lived* secrets at the time of terraform apply

Vault Injection via Data Source

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

A vault server is provisioned
A vault engine is configured e.g. AWS Secret Engine
Vault will create a machine user for AWS
Vault will generate short-live AWS credentials from that machine user
Vault will manage and apply the AWS Policy

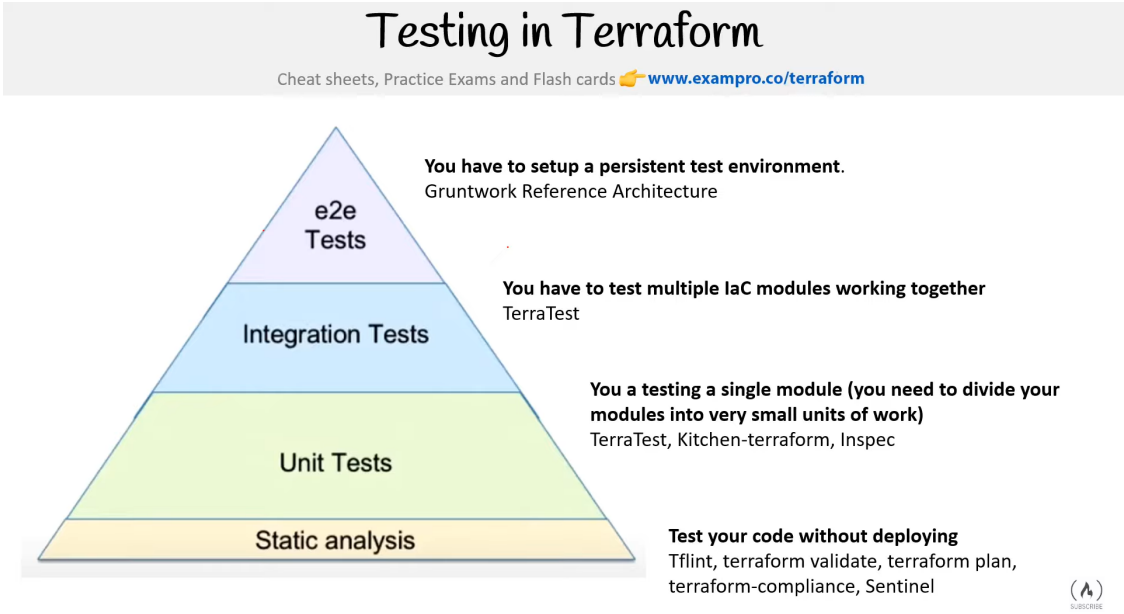
Within our Terraform we can provide a Data Source to Vault

```
data "vault_aws_access_credentials" "creds" {
  backend = data.terraform_remote_state.admin.outputs.backend
  role    = data.terraform_remote_state.admin.outputs.role
}

provider "aws" {
  region     = var.region
  access_key = data.vault_aws_access_credentials.creds.access_key
  secret_key = data.vault_aws_access_credentials.creds.secret_key
}
```

When Terraform Apply is run, it will pull short –lived credentials to be used scope for the duration of the current run.

Everytime you run apply you will get new short-lived credentials.

▼ Testing in Terraform



▼ Miscellaneous

- Terraform will store the token in plain text in the following file for use by subsequent commands: `/Users/redacted/.terraform.d/credentials.tfrc`.
- For local state, Terraform stores the workspace states in a directory called `terraform.tfstate.d`. This directory should be treated similarly to local-only `terraform.tfstate`.
- The Terraform local backend stores its state in a file called "terraform.tfstate" in the current working directory.

▼ Read and write configuration

- Terraform uses its own configuration language to determine the goal state for the infrastructure it manages.

▼ The below resources describe some of the features of Terraform's configuration language.

▼ Resources describe infrastructure objects

- **How Terraform Applies a Configuration**
  - When Terraform creates a new infrastructure object represented by a `resource` block, the identifier for that real object is saved in Terraform's state, allowing it to be updated and destroyed in response to future changes. For resource blocks that already have an associated infrastructure object in the state, Terraform compares the actual configuration of the object with the arguments given in the configuration and, if necessary, updates the object to match the configuration.
- **Accessing Resource Attributes**
  - Use the `<RESOURCE TYPE>.<NAME>.<ATTRIBUTE>` syntax to reference a resource attribute in an expression.
  - In addition to arguments specified in the configuration, resources often provide read-only attributes with information obtained from the remote API; this often includes things that can't be known until the resource is created, like the resource's unique random ID.
  - Many providers also include data sources, which are a special type of resource used only for looking up information.
- **Resource Dependencies**
  - Some resources must be processed only after other resources due to dependency this is because of how the terraform works.
  - Most resource dependencies are handled automatically. Terraform analyses any expressions within a `resource` block to find references to other objects, and treats those references as implicit ordering requirements when creating, updating, or destroying resources, It is not necessary to manually specify dependencies between resources.
  - some dependencies cannot be recognized implicitly in configuration, in this rare areas defining dependencies explicitly is required.
- **Local-only Resources**
  - local-only resource types exist for generating private keys, issuing self-signed TLS certificates, and even generating random ids. While these resource types often have a more marginal purpose than those managing "real" infrastructure objects.
  - The behavior of local-only resources is the same as all other resources, but their result data exists only within the Terraform state. "Destroying" such a resource means only removing it from the state and discarding its data.
- The Meta-arguments of resource including `depends_on`, `count`, `for_each`, `provider`, and `lifecycle`.

- Provisioners → configuring post-creation actions for a resource using the `provisioner` and `connection` blocks.

▼ Data sources let Terraform fetch and compute data

- *`Data sources`* allow Terraform to use information defined outside of Terraform, defined by another separate Terraform configuration, or modified by functions.
- **Using Data Sources**
  - A data source is accessed via a special kind of resource known as a *data resource*.

```
// declared using a data block:

data "aws_ami" "example" {
  most_recent = true

  owners = ["self"]
  tags = {
    Name   = "app-server"
    Tested = "true"
  }
}
```

- Query Data Sources tutorial guides you through using data sources

▼ Resource addressing lets you refer to specific resources

- A *`resource address`* is a string that identifies zero or more resource instances in your overall configuration.

```
[module path][resource spec]
```

- An example of the `module` keyword delineating between two modules that have multiple instances.

```
module.foo[0].module.bar["a"]
```

- **Resource spec → A resource spec addresses a specific resource instance in the selected module.**

```
resource_type.resource_name[instance index]
```

- Named values let you reference values
- Create Resource Dependencies tutorial guides you through managing related infrastructure using implicit and explicit dependencies
- Terraforms Resource Graph ensures proper order of operations

▼ Complex types let you validate user-provided values

- **Type Constraints**

- Terraform module authors and provider developers can use detailed type constraints to validate user-provided values for their input variables and resource arguments.

- **Type Keywords and Constructors**
  - Type constraints are expressed using a mixture of `type keywords` and function-like constructs called `type constructors`.
  - Type keywords are unquoted symbols that represent a static type.
  - Type constructors are unquoted symbols followed by a pair of parentheses, which contain an argument that specifies more information about the type.
- **Primitive Types**
  - A `primitive type` is a simple type that isn't made from any other types.
  - string, bool & number.
- **Conversion of Primitive Types**
  - The Terraform language will automatically convert `number` and `bool` values to `string` values when needed, and vice-versa as long as the string contains a valid representation of a number or boolean value.

▼ **Complex Types**

- A `complex type` is a type that groups multiple values into a single value.
- There are two categories of complex types: `collection types` (grouping similar values) and `structural types` (grouping dissimilar values).
  - A `collection type` allows multiple values of *one* other type to be grouped together as a single value.
  - The three kinds of `collection type` in the Terraform language are:
    - `list(...)`: a sequence of values identified by consecutive whole numbers starting with zero.
    - `map(...)`: a collection of values where each is identified by a string label.
    - `set(...)`: a collection of unique values that do not have any secondary identifiers or ordering.
  - A `structural type` allows multiple values of *several distinct types* to be grouped together as a single value.
    - `Structural types` require a `schema as an argument`, to specify which types are allowed for which elements.
    - The two kinds of `structural type` in the Terraform language are:
      - `object(...)`: a collection of named attributes that each have their own type.
        - The schema for object types is `{ <KEY> = <TYPE>, <KEY> = <TYPE>, ... }` — a pair of curly braces containing a comma-separated series of `<KEY> = <TYPE>` pairs.

```
// an object type of object({ name=string, age=number })

{
  name = "John"
  age  = 52
}
```

- `tuple(...)`: a sequence of elements identified by consecutive whole numbers starting with zero, where each element has its own type.
  - The schema for tuple types is `[<TYPE>, <TYPE>, ...]` — a pair of square brackets containing a comma-separated series of types.

```
// a tuple type of tuple([string, number, bool])

["a", 15, true]
```

- **Dynamic Types: The "any" Constraint**
  - The keyword `any` is a special construct that serves as a placeholder for a type yet to be decided.
  - `any` is not *itself* a type.
  - Terraform will attempt to find a single actual type that could replace the `any` keyword to produce a valid result.

  ▼ example

  If the given value were `["a", "b", "c"]` -- whose physical type is `tuple([string, string, string])`, Terraform analyzes this as follows:

  - Tuple types and list types are *similar* per the previous section, so the tuple-to-list conversion rule applies.
  - All of the elements in the tuple are strings, so the type constraint `string` would be valid for all of the list elements.
  - Therefore in this case the `any` argument is replaced with `string`, and the final concrete value type is `list(string)`.
  - If the given value were instead `["a", 1, "b"]` then Terraform would still select `list(string)`, because of the primitive type conversion rules, and the resulting value would be `["a", "1", "b"]` due to the string conversion implied by that type constraint.
  - If the given value were instead `["a", [], "b"]` then the value cannot conform to the type constraint: there is no single type that both a string and an empty tuple can convert to. Terraform would reject this value, complaining that all elements must have the same type.
- **Optional Object Type Attributes**
  - Terraform typically returns an error when it does not receive a value for specified object attributes.

- When you mark an attribute as optional, Terraform instead inserts a default value for the missing attribute.

```
variable "with_optional_attribute" {
  type = object({
    a = string                # a required attribute
    b = optional(string)      # an optional attribute
    c = optional(number, 127) # an optional attribute with default value
  })
}
```

▼ The `optional` modifier takes one or two arguments.

- **Type:** (Required) The first argument specifies the type of the attribute.
- **Default:** (Optional) The second argument defines the default value that Terraform should use if the attribute is not present. This must be compatible with the attribute type. If not specified, Terraform uses a `null` value of the appropriate type as the default.

▼ Built in functions help transform and combine values

- The Terraform language does not support user-defined functions, and so only the functions built in to the language are available for use.
- While the configuration language is not a programming language, you can use several built-in functions to perform operations dynamically.

▼ Perform Dynamic Operations with Functions tutorial walks you through using Terraform functions

- use the `templatefile` function to dynamically create an EC2 instance user data script.
- use the `lookup` function to reference values from a map.
- use the `file` function to read the contents of a file.

▼ Create Dynamic Expressions tutorial shows you how to create more complex expressions

- The Terraform configuration language supports complex expressions to allow you to compute or generate values for your infrastructure configuration.
- Expressions can be simple string or integer values, or more complex values to make your configuration more dynamic.

▼ **Using `Conditional Expressions`.**

- A *`conditional expression`* uses the value of a boolean expression to select one of two values.

```
condition ? true_val : false_val

// If condition is true then the result is true_val. If condition is false then the
 result is false_val.
```

▼ **Local Block**

- 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.
- **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 a `splat` expression**

- The `splat` expression captures all objects in a list that share an attribute.
- The special `*` symbol iterates over all of the elements of a given list and returns information based on the shared attribute you define.

```
output "private_addresses" {
  description = "Private DNS for AWS instances"
  value       = aws_instance.ubuntu[*].private_dns
}
```

▼ Dynamic blocks allow you to construct nested expressions within certain configuration blocks

▼ **`dynamic` Blocks**

- You can dynamically construct repeatable nested blocks like `setting` using a special `dynamic` block type, which is supported inside `resource`, `data`, `provider`, and `provisioner` blocks.

```
resource "aws_elastic_beanstalk_environment" "tfenvtest" {
  name                = "tf-test-name"
  application         = "${aws_elastic_beanstalk_application.tftest.name}"
  solution_stack_name = "64bit Amazon Linux 2018.03 v2.11.4 running Go 1.12.6"

  dynamic "setting" {
    for_each = var.settings
    content {
      namespace = setting.value["namespace"]
      name = setting.value["name"]
      value = setting.value["value"]
    }
  }
}
```

- A `dynamic` block can only generate arguments that belong to the resource type, data source, provider or provisioner being configured. It is *not* possible to generate meta-argument blocks such as `lifecycle` and `provisioner` blocks, since Terraform must process these before it is safe to evaluate expressions.
- Overuse of `dynamic` blocks can make configuration hard to read and maintain, so we recommend using them only when you need to hide details in order to build a clean user interface for a re-usable module. Always write nested blocks out literally where possible.

- Because Terraform manages your infrastructure, it sometimes needs access to sensitive data. You can inject sensitive data into Terraform configuration using Vault.

- Inject secrets into Terraform using the Vault provider
- Vault Provider for Terraform

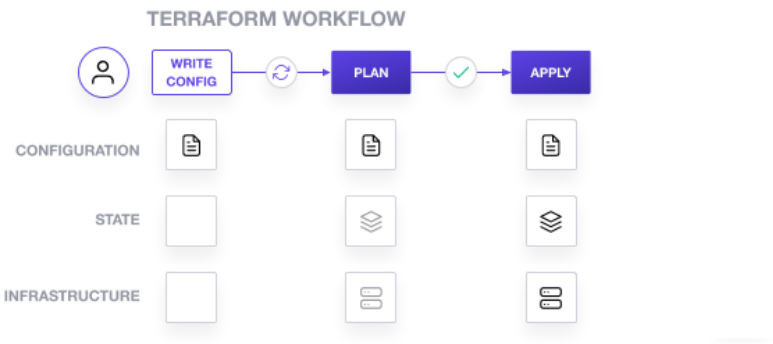
▼ Import Terraform Configuration

> reference: https://developer.hashicorp.com/terraform/tutorials/state/state-import#import-the-container-into-terraform

- Terraform supports bringing the existing infrastructure under its management by safely & consistently manage the infrastructure using a predictable workflow.

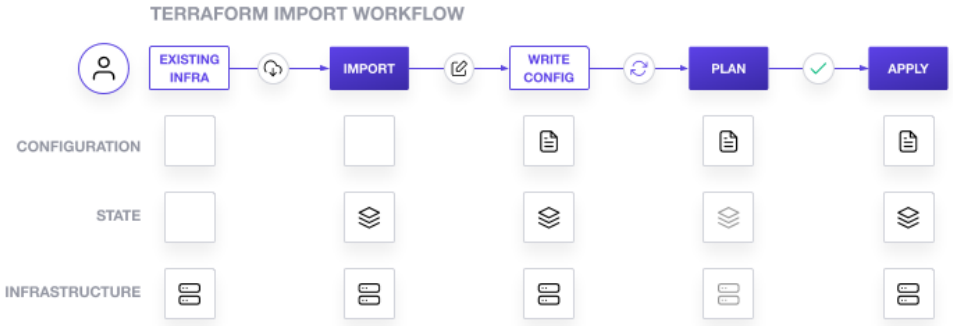
▼ When create new infrastructure with Terraform, you usually use the following workflow

1. Write Terraform configuration that defines the infrastructure you want to create.
2. Review the Terraform plan to ensure the configuration will result in the expected infrastructure.
3. Apply the configuration to have Terraform create your infrastructure.



▼ Importing infrastructure involves five steps

1. Identify the existing infrastructure you will import.
2. Import infrastructure into your Terraform state file.
3. Write Terraform configuration that matches that infrastructure.
4. Review the Terraform plan to ensure the configuration matches the expected state and infrastructure.
5. Apply the configuration to update your Terraform state.



- After importing infrastructure into state file & creating empty config file.
- There are two approaches you can use to update the configuration
  - Using current state file
  - Cherry-pick configuration

▼ Refactor Monolithic Terraform Configuration

- Some Terraform projects start as a `monolith`, a Terraform project managed by a single main configuration file in a single directory, with a single state file, suitable for small projects.
- As Infrastructure grows, restructuring your monolith into logical units will make your Terraform configurations less confusing and safer to manage.

▼ Separate states

- State separation signals more mature usage of Terraform; with additional maturity comes additional complexity.
- There are two primary methods to separate state between environments: directories and workspaces.
- Use a directory structure, To separate envs. with potential configuration differences.
- Use workspaces for environments that do not greatly deviate from one another, to avoid duplicating your configurations.

▼ Using Workspace Env.

- creating workspaces along with creating `.tfvars` files allows to enable separate env variables wrt separate workspaces.
- `terraform.tfstate.d` this folder will contain the `terraform.tfstate` state files when working with workspaces

▼ State storage in workspaces

- When you use the default workspace with the local backend,
  your `terraform.tfstate` file is stored in the root directory of your Terraform project.

- When you add additional workspaces your state location changes; Terraform internals manage and store state files in the directory `terraform.tfstate.d`.

▼ `terraform workspace [options]`

▼ terraform workspace list

- by default, default workspace will be present & it cannot be deleted.

▼ terraform workspace new [NAME]

```
terraform workspace new dev
```

▼ terraform workspace delete [NAME]

```
terraform workspace delete dev
```

▼ terraform workspace select

```
terraform workspace select prod
```

- terraform destroy -var-file="dev.tfvars" -auto-approve

▼ **State storage in workspaces**

When you use the default workspace with the local backend, your `terraform.tfstate` file is stored in the root directory of your Terraform project. When you add additional workspaces your state location changes; Terraform internals manage and store state files in the directory `terraform.tfstate.d`.

▼ Lock and Upgrade Provider Versions

- . `terraform.lock.hcl` file keeps all modules & providers version, constraints & hashes.
- Initializing a terraform config for the 1st time with terraform v1.1 or later, terraform will generate a new `.terraform.lock.hcl` file in current working directory.
- This file should be included in version control repositories to ensure that terraform uses the same provider versions across teams & other environments.
- **Upgrade the AWS provider version:**

```
terraform init -upgrade
```

  - The `-upgrade` flag will upgrade all providers to the latest version consistent within the version constraints specified in your configuration.
  - We can also downgrade the versions by using `-upgrade` flag.

⚠️ **Note:**
- You should never directly modify the lock file.
