# Terraform Study Guide 2.0

Breakdown by content area:

**1.0 Understand infrastructure as code (IaC) concepts**

**2.0 Understand Terraform's purpose (vs other IaC)**

3.0 Understand Terraform basics

4.0 Use the Terraform CLI (outside of core workflow)

5.0 Interact with Terraform modules

6.0 Navigate Terraform workflow

7.0 Implement and maintain state

8.0 Read, generate, and modify configuration

9.0 Understand Terraform Cloud and Enterprise capabilities

# Terraform Basics

* **What is infrastructure as code?** It’s infrastructure (CPUs, memory, disk, firewalls, etc) defined as code within definition files/ managing your IT infrastructure using configuration files.
* **Why is it important?** It's really the versioning of it, the reusability of the code, and the ability to then do automation on top of it.
* **The infrastructure Terraform can manage** includes low-level components such as compute instances, storage, and networking, as well as high-level components such as DNS entries, SaaS features, etc.
* **“Day 0”** code provisions and configures your initial infrastructure.
* **“Day 1”** refers to OS and application configurations you apply after you’ve initially built your infrastructure. Includes OS updates, patches, app config. Example(downloading Python3 onto an ec2 instance)

**Key features of terraform:**

1. **IAC**: allows a blueprint of your datacenter to be versioned and treated as you would any other code. Additionally, infrastructure can be shared and re-used.
2. **Execution Plans**: Terraform Plan “Planning” step where it generates an execution plan which shows what Terraform will do when you apply.
3. **Resource Graph**: builds a graph of all your resources and parallelizes the creation and modification of any non-dependent resources.
4. **Change automation**: Complex changesets can be applied to your infrastructure with minimal human interaction.

# Block Types

|  |  |
| --- | --- |
| Resource Block | Each resource block describes one or more infrastructure objects, such as virtual networks, compute instances, or higher-level components such as DNS records. |
| Data Source Blocks | A data source block is tracking data for an already existing resource. |
| Provider Block |  |
| Provisioner Block |  |
| Terraform Block | Includes the required providers terraform will use to provision your infrastructure |

* The format of a block configuration is as follows:

“<block type>” “<resource type>" " <local name>

**\*\* Reserved words are invalid resource names**

# Terraform Configuration Block

* Includes the required providers terraform will use to provision your infrastructure

**Example of a terraform configuration Block**

Graphical user interface, text, application, chat or text message

Description automatically generated

* A special configuration block for controlling terraforms own behavior.
* This block only allows constant values, named resources and variables are not allowed in it.
* **Examples include:** 
  + - Configuring backend for storing state files
    - Specifying a required TF version
    - Specifying a required Terraform provider version and its requirements
    - Enable and test Terraform experimental features
    - Passing Metadata to providers in modules
* With a partial configuration, the remaining configuration arguments must be provided as part of [the initialization process](https://www.terraform.io/docs/backends/init.html#backend-initialization). Ways to supply the remaining arguments:

1. **Interactively:** Terraform will interactively ask you for the required values unless interactive input is disabled.
2. **File**: A configuration file may be specified via the init command line. To specify a file, use the -backend-config=PATH option when running terraform init. If the file contains secrets it may be kept in a secure data store, such as Vault, in which case it must be downloaded to the local disk before running Terraform.
3. **Command-line key/value pairs:** Key/value pairs can be specified via the init command line. Note that many shells retain command-line flags in a history file, so this isn't recommended for secrets. To specify a single key/value pair, use the -backend-config="KEY=VALUE" option when running terraform init

# Dynamic Blocks

* You can dynamically construct repeatable nested blocks like ingress using a special dynamic block type, which is supported inside resource, data, provider, and provisioner blocks.
* A dynamic block acts much like a for expression but produces nested blocks instead of a complex typed value. It iterates over a given complex value and generates a nested block for each element of that complex value.
* A dynamic block acts like a for expression

**Example of a Dynamic Block**

Text

Description automatically generated

# Supported Version Control System

* GitHub.com (must be used for public modules)
* GitHub.com (OAuth)
* Github.com Enterprise
* GitLab.com
* GitLab EE and CE
* Bitbucket Cloud
* Bitbucket Server
* Azure DevOps Server
* Azure DevOps Services

# Downloading Terraform

**Method 1:**

1. download zipped binary from Hashi corp website.
2. Unzip TF Binary
3. As a best practice place, it on your systems $ Path

**Method 2:**

* set up a HashiCorp Tf repo on to nux (Debian, RHEL, Amazon Linux)

1. use package manager to install terraform
2. Package installs and setup so it’s ready to use Providers

# Workflow Steps

1. Write: author iac
2. Plan: preview changes before applying
3. Apply: provision reproducible infrastructure

# Terraform Providers

* **Terraform Providers:** Terraforms way of **abstracting** integrations with **API control layer** of the infrastructure vendors.
* you should constrain the acceptable provider versions via configuration file to ensure that new versions with breaking changes will not be automatically installed by terraform init in the future. Required\_providers
* by default, terraform looks for providers in the **terraform provider registry**
* Providers are **plug-ins** they are released on a separate rhythm from terraform itself and each provider has its own series of version numbers.
* provider blocks are not required! If the block would otherwise be empty you do not need one
* A provider is responsible for understanding API interactions and exposing resources. Providers generally are an IaaS.
* Terraform provider block has two meta-arguments available:

1. Alias: which is for using the same provider with different configurations for different resources.
2. Version: which is no longer recommended (use provider requirements instead which takes a version value) \*\* **see terraform configuration block for details**

* As best Practice providers should be pegged down to a specific version, so that any charges across provider version doesn’t break the terraform code.
* If the provider version is not provided the latest is downloaded.
* When you have multiple configurations for one provider use the alias meta-argument to provide an extra name segment

\*\* You can write your own providers. Which are typically written in GO using the terraform plug in SDK Terraform finds and installs Providers when initializing working directories (via terraform init)

**Graphical user interface, text, application

Description automatically generatedExample of a provider block**

# Terraform Plugins

* written in Go
* executable binaries executed as a separate process and communicate with the main Terraform binary over an RPC interface
* The ~/.terraform.d/plugins only for the third-party plugins
* Terraform init creates a directory in the current directory called **.terraform/plugins** (NOT INCLUDING THIRD PARTY)
* Providers are currently the only plugin type most Terraform users will interact with. Terraform also supports third-party provisioner plugins, but we discourage their use.
* Third Party Plugins are stored in **~/.terraform.d/plugins** on Unix systems or %APPDATA%\terraform.d\plugins on Windows.
* Terraform Providers contain all the code needed to authenticate and connect to a service on behalf of the user

# Variables

* Variables make configuration reusable
* Best practice is putting variables in a terraform.tfvars file or variable.tf
* Config parameter sensitive prevents terraform from showing the value of the tf variable during the run which is a bool value default false.
* Variable validation allows you to set criteria for allowed values for a variable. Created in tf version 0.13
* \*\* terraform.tfstate && terraform.tfvars should not be committed to GitHub do a git ignore. Terraform.tfvars may contain sensitive data, such as passwords or IP addresses of an environment that you may not want to share with others.

**Example of variable validation**

**Text

Description automatically generated**

# Multi Cloud and Provider-agnostic Benefits

* Multi-cloud deployment increases fault tolerance. This means in the event of failure there is a more graceful recovery of a region or provider.
* The benefits of being provider-agnostic means there can be a **single configuration that manages many providers.**

Terraform Type Constraints

## Complex Types

* A type that groups multiple values into a single value. Complex types are represented by type constructors, but several of them also have shorthand keyword versions.

|  |  |
| --- | --- |
| List | [], **list(string),** list(any) accepts any element type as long as every element is the same type. **This is for compatibility with older configurations; for new code, we recommend using the full form.** |
| Set | **toset(["a", "b", 3])** a collection of unique values that do not have any secondary identifiers or ordering. |
| Map | Maps can be made with braces ({}) and colons (:) or equals signs (=): **{ "foo": "bar", "bar": "baz" } OR { foo = "bar", bar = "baz" }.** Quotes may be omitted on keys, unless the key starts with a number |

## Structural Types

* allows multiple values of several distinct types to be grouped together as a single value.

|  |  |
| --- | --- |
| Object | **{ <KEY> = <TYPE>, <KEY> = <TYPE>, ... }** — a pair of curly braces containing a comma-separated series of <KEY> = <TYPE>. object({ name=string, age=number }) |
| Tuple | a sequence of elements identified by consecutive whole numbers starting with zero, where each element has its own type  **tuple([string, number, bool])** an example |

## Primitive Types

* simple type that isn't made from any other types. All primitive types in Terraform are represented by a type keyword.

|  |  |
| --- | --- |
| String | A sequence of Unicode characters representing some text, such as "hello" |
| Number | a numeric value. The number type can represent both whole numbers like 15 and fractional values such as 6.283185. |
| Bool | either true or false. bool values can be used in conditional logic. |

# Terraform Provisioners

* Way of bootstrapping Custom scripts, commands or action.
* can be run either locally (on the same system where terraform commands are issued from) or remotely on resources spun up through the terraform deployment
* **each individual resource can have its own "Provisioner**"
* **remote exec provisioner** supports both **ssh** and **winrm** type connections.
* There are 2 types of provisioners: "creation-time(when resource created) and "Destroy time”(when resource is destroyed)
* **terraform cannot track changes to Provisioners** as they can take any independent action, hence they are not tracked by Terraform state files
* Provisioners are recommended for use when you want to invoke actions not covered by terraform declarative model.
* if the command within a **Provisioner returns nonzero code**, it’s considered failed and the underlying resource is **tainted**.
* **provisioners are not tracked in the state file.**
* ***Provisioners should only be a last resort!!!***

**Example of local-exec provisioner that will run the command on local machine**

**Text

Description automatically generated**

# Terraform State

* Terraform creates a state file when a project is first initialized (**Terraform init**). Terraform uses this local state to create plans and make changes to your infrastructure.
* Resource tracking
* A way for terraform to keep tabs on What has been deployed.
* Critical to terraform functionality
* Terraform stores the state in a file in the current working directory where Terraform was run
* Json dump containing all metadata about terraform deployment
* Stored in Flat files, by default named "terraform.tfstate”
* Helps TF calculate deployment delta and create new deployment plans.
* never lose your TF state file
* stores sensitive data in plain text

**Without tf state terraform would not work.**

* **maps real-world resources** to terraform configuration.
* **by default,** state is stored locally in a file called terraform.tf state
* **Prior to any modification operation**, terraform refreshes the state file
* **Resource dependency metadata** is also tracked via the state file which helps boost deployment Performance by caching resource attributes for subsequent use
* terraform state

**Commands allows you to manipulate and read the tf state file**

Table

Description automatically generated with medium confidence

**terraform state show** only shows the one resource you passed during the command **terraform state show <resource>**

* **terraform state rm <resource>** unmanages a resource while  
  terraform destroy will destroy those resources

# Local & Remote state

**Local State Storage:** Saves tf state on local system (terraform.tfstate)

* typically used for individual projects & testing

**Remote State Storage:** Saves state to a remote datasource. optional. Examples of storage include AWS S3, Google storage  
advantages:

1. Allows shared state files between distributed teams
2. Better security & availability on the cloud / better back up

**State locking: locks state** **so parallel executions don't coincide**

local backend auto state locks

If state locking fails, terraform will not continue. You can disable state locking for most commands with the -lock flag but it is not recommended.

**Remote backends that support state locking:**

1. AWS S3
2. Google cloud storage
3. Hashi Corp

if the state file is stored remotely it enables sharing output Values with other TF configuration or code.

A lock can be forced open with force-unlock which requires a unique nonce lock ID

**terraform init:** will make sure the right provider is pulled down, makes sure the bucket exists and makes sure the right version is installed/used.

# Terraform Modules

**Terraform modules**: another folder or collection of Terraform code files. You reference the outputs of that code into other parts of the terraform project.

A **module** is a container for multiple resources used together

Every terraform configuration has at least one module, Called the **root module**, which consist of code files in your main working directory

* **Published modules are managed via Git and GitHub.** publishing a module takes only a few minutes. Once a module is published, you can release a new version of a module by simply pushing a properly formed Git tag.
* A module that is called by another configuration is sometimes referred to as a "child module" of that configuration.
* child modules have variables set in the configuration of the parent module
* root module variables can be set with CLI and environment variables.
* Version constraints are supported only for modules installed from a module registry, such as the public [Terraform Registry](https://registry.terraform.io/) or [Terraform Cloud's private module registry](https://www.terraform.io/docs/cloud/registry/index.html)
* Benefits of Published modules via the Terraform Registry

1. Automatically generated documentation
2. Allow browsing version histories
3. Support Versioning
4. Show examples and READMEs

* A Terraform module (usually the root module of a configuration) can call other modules to include their resources into the configuration. A module that has been called by another module is often referred to as a **child module**.
* **Child modules can be called multiple** **times** within the same configuration, and multiple configurations can use the same child module.

**modules can be downloaded or referenced from:**

* **Terraform public registry:** All publicly available modules terraform downloads them and places them in a directory on your system usually hidden
* **A private registry:** you would want to use for closed source code or security reasons
* **Your local system:** save them in a local folder on your system and reference that folder using its path

**Other allowed Params for modules include:**

1. **Count**
2. **For\_each**
3. **Providers**
4. **Depends\_on**

\*\* Always have a version in your module block to avoid unwanted effects to your deployments

Diagram

Description automatically generated

# Terraform Built-in Functions

* Terraform comes prepackaged with functions to help you transform and combine values
* User defined functions are not allowed
* General syntax function\_name(arg1,arg2)
* Built in functions make TF code dynamic and flexible

List of functions

* File: inserting files into resources
* Max: for determining the max integer value from a provided list
* Flatten: create a singular list out of provided set of lists
* Zip map constructs a map from a list of keys and a corresponding list of values. A map is denoted by {} whereas a list is denoted by []
* Timestamp () will give the timestamp
* Join

Text

Description automatically generated

# Debugging Terraform

* **Tf\_log** an env variable for enabling verbose logging in Terraform. By default, **it will send logs to stderr (standard error output).**
* **Can be set to the following levels:** TRACE, DEBUG, INFO, WARN, ERROR
* **TRACE IS THE MOST VEROSE** level of logging and the most reliable
* **To persist logged output, use the TF\_Log\_Path environment variable**

# Setting Environment Variables

|  |  |
| --- | --- |
| **Setting the region** | **export TF\_VAR\_region=us-west-1** |
| **Setting logging variables**  **(**TRACE, DEBUG, INFO, WARN, ERROR**)** | **export TF\_LOG=TRACE(trace is the most verbose)** |
| **Setting logs output variables**  (where the log should persist its output) | **export TF\_LOG\_PATH= ./terraform.log** |
| If set to "false" or "0", causes terraform commands to behave as if the -input=false flag was specified (disables prompts for variables) | **export TF\_INPUT** |
| **Environment variables must be in this format** | **export TF\_VAR\_name** |
| will only affect that command listed | **export** TF\_CLI\_ARGS\_name |
| affects all Terraform commands | **export TF\_CLI\_ARGS** |
| changes the location where Terraform keeps its per-working-directory data | **export TF\_DATA\_DIR** |
| For multi-environment deployment, in order to select a workspace, instead of doing terraform workspace select your\_workspace | **export TF\_WORKSPACE** |
| Terraform will output debug messages to display ignored files and folders if set to trace. (useful when debugging large repos) | **export TF\_IGNORE = trace** |

# Terraform cloud workspaces (use these on team)

* Directories hosted in Terraform cloud
* Think of workspaces as distinct deployments hosted in the cloud where you don’t need to worry about segregation storage.
* Stores old versions of state filed by default
* Maintains a record of all execution activity which allows for auditing and investigating deployments more readily and easily
* All TF commands executed on managed TF cloud VMs

# Open Source vs. Cloud

* Terraform OSS workspaces store alternate state files in the same working directory. It does that by creating a separate directory within main terraform directory called the terraform.tfstate.d directory where it stores a folder for each state file tracked by the different workspaces.
* (OSS) A workspace can only be configured to a single Version control system (VCS) repo. however, multiple workspaces can use the same repo, if needed.
* Backends that support multiple workspaces are Azure-RM, Consul, COS,GCS, Kubernetes, Local, Manta, Postgres, Remote, S3
* Terraform OSS: Workspaces generate and track different state files against the same terraform code in a directory, hosted locally on your system.
* Terraform Cloud: workspaces are hosted in the cloud and track configuration, variables, state, and secrets and can be interacted with programmatically

Table

Description automatically generated

# Terraform Cloud vs. Enterprise

|  |  |  |  |
| --- | --- | --- | --- |
| **Terraform Cloud** | **Both Cloud and Enterprise** | **Terraform Enterprise** | **Terraform OSS** |
| Cloud Operations | Audit Logging | Private Operations | Local CLI operations |
| Self-hosted Agents | Self-service Infrastructure | Clustering(removed Jan 15 2021) | Community |
|  | SAML/SSO | Locally Hosted Installation |  |
|  | Concurrent runs |  |  |
|  | Private Module registry (**free but have to use cloud and not OSS**) |  |  |
|  | Sentinel Policies |  |  |
|  | Cost Estimation |  |  |

# Terraform Cloud and Enterprise offerings

* **Hashicorp Sentinel** is a framework which enforces adherence to policies within your terraform code. In other words, it is code that enforces restrictions on your terraform code.
* Sentinel has its own language to write policies. It is human redable and designed to be approachable by non-programmers
* Policies are checked when a run is performed, after the terraform plan but before it can be confirmed or the terraform apply is executed.
* Written using the sentinel language

**Benefits**

* **Sandboxing-** Guardrails for automation
* **Codification:** Easier understanding, better collaboration
* **Version Control**
* **Testing && Automation**

**Use Cases**

* Enforce CIS standards across AWS accounts
* Checking to make sure only t3. micro instance types are used for EC2
* Ensuring Security Groups do not allow traffic to port 22

**Sentinel policies** are written in sentinel

Example of a Sentinel Policy

Text

Description automatically generated

# Terraform Registry

* A repository of publicly available Terraform providers and modules
* Find documentation and can contribute
* You can publish and share your own modules, too
* You can collaborate with contributors to make changes to providers and modules
* Can be directly referenced in your Terraform code
* Registry.terraform.io

# Terraform Vault

* Secrets management software
* **Avoid putting secret or sensitive variables in config or state files.**
* Set secret variables for provider config block in environment variables
* Dynamically provisions credential and rotates them
* Encrypts sensitive data in transit and at rest and provides fine-grained access to secrets using ACLs

Benefits

* Developers don’t need to manage long lived credentials
* [Terraform's Vault provider](https://www.terraform.io/docs/providers/vault/) to generate appropriately scoped & short-lived AWS credentials to be used by Terraform to provision resources in AWS
* Inject secrets into your terraform deployment at runtime
* Fine grained ACLs for access to temporary credentials

# Important Commands

* **Terraform init:** Initializes downloaded and/or installed providers
* The following actions are performed during a terraform init:

1. Initializes the backend
2. Downloads the declared providers which are supported by Hashicorp
3. Initializes downloaded/installed providers

* **terraform init -upgrade:** to upgrade all modules
* **terraform fmt:** used for rewriting Terraform configuration files to a canonical format and style.
* **Terraform Taint:** Marks a resource as tainted, forcing it to be destroyed and recreated on the next apply.
  + modifies state file only.
  + After a resource is marked the next plan shows it will be destroyed and recreated on the next apply
  + Can cause other resources to be modified
  + Useful when we want a side effect of a recreation that is not visible in the attributes of the resource. For ex/rebooting the machine from a base image causing a new startup script to run.
  + Scenarios when you might use terraform taint:
    - To cause provisioners to run
    - Replace misbehaving resources forcefully
    - To mimic side effects of recreation not modeled by any attributes of the resource.
* **Terraform Import:** Imports existing resources into terraform. \*\*\* can only import resources into the state you are responsible for manually adding the resource to the configuration file first!!
* Maps existing resources to Terraform using an “ID”
* “ID” is dependent on the underlying vendor, for example to import an AWS EC2 instance you’ll need to provide its instance ID
* Importing the same resource to multiple Terraform resources can cause unknown behavior and is not recommended
* Scenarios when terraform import is used:
  + - When you need to work with existing resources
    - Not allowed to create new resources
    - When you’re not in control of creation process of infrastructure.
* [**terraform providers**](https://www.terraform.io/docs/cli/commands/providers.html) command to get information about the providers required by the current working directory's configuration.
* Use the[**terraform version**](https://www.terraform.io/docs/cli/commands/version.html) command (or terraform -version) to show the specific provider versions installed for the current working directory.
* Use the [**terraform providers schema**](https://www.terraform.io/docs/cli/commands/providers/schema.html) command to get machine-readable information about the resources and configuration options offered by each provider.
* **Terraform validate** validates the configuration files in a directory, referring only to the configuration and not accessing any remote services such as remote state, provider APIs, etc.
* **Terraform Plan** creates an execution plan and determines what changes are required to achieve the desired state in the configuration files.
* **Terraform Refresh:** used to reconcile the state Terraform knows about (via its state file) with the real-world infrastructure. This can be used to detect any drift from the last-known state, and to update the state file. This **does not modify infrastructure but does modify the state file**. If the state is changed, **this may cause changes to occur during the next plan or apply**.

**Terraform Workspace (CLI):**

* The terraform workspaces are alternate state files within the same working directory
* Creating different workspaces is useful to manage different stages of deployment (sandbox or production)
* **At first the backend only has one workspace 'default'. This workspace cannot be deleted.**
* Access to a workspace name is provided through the ${terraform. Workspace} variable
* **Backends that support multiple workspaces include** local, remote, AzureRM, Consul, COS (Tencent Cloud Object Storage), GCS (Google cloud storage), Kubernetes, Manta, Postgres, S3

**Scenarios when to use:**

* Test changes using a parallel, distinct copy of infrastructure.
* It can be modeled against branches in version control such as Git

**Commands:**

**terraform workspace new:** create a workspace

**terraform workspace select:** Select a workspace

**terraform workspace list**: list the available workspaces

* **Terraform state: used for advanced state management. Used to change state instead of changing state directly.**
* **Subcommands include:**

1. **List: list all resources in TF state fle**
2. **Mv:**
3. **Pull**
4. **Push**
5. **Rm**
6. **Show:** used to provide human-readable output from a state or plan file.

* **Terraform console** command provides an interactive console for evaluating expressions.
* The Terraform console allows you to experiment with Terraform interpolations.
* **Terraform Apply**: Applies the changes required in the target infrastructure in order to reach the desired config
* by default, **terraform provisions 10 resources** concurrently during a terraform apply. Use the parallelism=n argument
* when a resource is successfully created but eventually fails during provisioning. What happens to the resource? it is marked tainted

# Miscellaneous

Terraform is available for the following operating systems

1. windows
2. Mac Os
3. FreeBSD
4. Open BSD
5. Linux
6. Solaris

# Version Best Practices

* **“>=1.0” represents the minimum version that’s compatible with the configuration.**

**Table

Description automatically generated**

* **“~>” only patch releases within a specific minor release.** This should not be used for modules intended to be reused across configurations because more often it forces users of the module to update many modules simultaneously when performing routine upgrades. **Specify a minimum version. v0.14.x**
* >= 0.13.0: means the required version is greater than version 0.13.0
* **~**> 1.2.0 will match any non-beta version of the provider between> = 1.2.0 and 1.2.9

Terraform cli commands

* Terraform login –
* terraform workspace show chadiamond
* terraform workspace delete chadiamond
* terraform providers lock
* terraform output
* terraform graph is a DOT format

# Terraform CLI Configuration file

* on Windows the file must be named terraform.rc
* the file must be named .terraformrc
* plugin\_cache\_dir = "$HOME/.terraform.d/plugin-cache"
* disable\_checkpoint = true