HashiCorp Certififed Terraform Associate Last day revision

Notes prepared by Shrikant Lavhate
kerneltalks.com

Terraform is written in the GO programming language.

Blocks

Terraform block

Particular config block that controls the behaviour of terraform.
Only constant values can be used in this block.

Sample terraform block:

```
terraform {
  experiments = [example]
  required_providers {
    aws = {
      version = ">= 2.7.0"
      source = "hashicorp/aws"
    }
  }
}
```

Providers block

Terraform relies on plugins called "providers" to interact with remote systems.

Sample provider block:

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

Two meta arguments defined and available by default for all providers blocks are :

1. `alias` - Use the same provider with diff configs for diff resources

2. `version` - no longer recommended. Use `provider requirements` instead.

Sample provider block wtih alias and its use in resources block :

```
provider "aws" {
  region = "us-east-1"
}
provider "aws" {
  alias  = "west"
  region = "us-west-2"
}
resource "aws_instance" "foo" {
  provider = aws.west
}
```

IBM Cloud does not have Hashicorp maintained provider.

Resources block

Defines components of your infrastructure.

Sample resources block:

```
resource "aws_s3_bucket" "tf-kerneltalks" {
  bucket = "tf-kerneltalks-98346"
  acl    = "private"
}
```

The resource follows by 2 entities.

1. Resource type. It always starts with the provider. So by looking at resource type you can determine which provides its using.
2. Logical resource name to be referred within your code. It's like a resource name in CloudFormation.

The actual block contains all the mandatory attributes for a selected resource type to create.

Outputs block

`sensitive = true` tf will not show these values in plan/apply output but those are still recorded in plain text in the state file.

Provisioner block

Provisioners can be used to model specific actions on the local machine or on a remote machine in order to prepare servers or other infrastructure objects for service. This block exists inside the resource block. Multiple provisioners can be defined.

They are not recommended to use. You can explorer other options like user-data scripts (aws bootstrap scripts) or cloud-init scripts before going to provisioners.
Remote-exec provisioner supports SSH for Linux and winrm (windows remote management) for Windows. They run on the remote resource created by terraform
Local-exec runs on the same machine where terraform is running. Terraform recommends to run local-exec against null resources.
There are 2 types of provisioners:

1. **Create time provisioner**: If not specified then by default all provisioners are create time. They get executed at resource creation. If it fails then create a process that taints the resource so that it will be recreated in the next `apply`.
2. **Destroy time provisioner**: They are specified by declaring `when = “destroy”` in the provisioner block. If it fails, the resource won’t be deleted by terraform.

You can specify `on_failure = “continue”` so that terraform execution continues even on provisioner failure.

Provisioner refers to its parent resource by “`self`” and not by the resource name/id.

Data block

Holds special resource named as data sources. It enables terraform to use data defined outside of terraform or modified by functions.

```
data "aws_ami" "example" {
  most_recent = true

  owners = ["self"]
}
```

Dynamic Block

Overuse of dynamic blocks is not recommended. IT makes code hard to understand and debug in case of issues.

Files

State File

`terraform.tfstate` & `terraform.tfstate.backup`

Created after first apply command and backup created once the current tfstate updates from subsequent apply command.

State file maintains the real world. So e.g. main.tf has your infra config and main.tfstate has real-world data of your infra. It's a mapping of config to real-world resource identities.

It also helps terraform to understand the resource ordering schematics. So in case, you delete a resource in main.tf, it determines deleting dependencies and ordering from state file since it can't determine it from main.tf as code is altered.

It also helps in performance boost since you don't need to sync resource attributes from the cloud every time. Some cloud providers don't allow parallel query api or api throttling impacts syncing every time you run plan/apply.

Remote state files recommended so that they can be leveraged by everyone working on the same code. It will have a lock so not all users can run apply at once and it also ensures every run begins with an updated/synced state file.

Lock file

`.terraform.lock.hcl`

Created during the `terraform init` command.
it is a lock file for various items that Terraform caches in the `.terraform` subdirectory of your working directory.
Use `force-unlock` to override the lock. It is to be used only when automatic unlocking fails. Because when it fails you will be supplied with a lock Id. This lock ID is required by `force-unlock` to proceed.

Terraform directory

`.terraform` (directory)

Contains providers directory

Variables files

```
terraform.tfvars
terraform.tfvars.json
*.auto.tfvars
```

`*.auto.tfvars.json`

Define variables to be used in the project.
Variables are evaluated in the below order. The later value takes precedence over the previous one.

1. Environment variable (only supports strings)
2. `terraform.tfvars`
3. `terraform.tfvars.json`
4. `*.auto.tfvars` (in alphabetical order)
5. `*.auto.tfvars.json` (in alphabetical order)
6. `-var` and `-var-file` arguments passed with command.

Types of variables:

- String (“hello” “test”)
- Number (1 2 3 ..)
- Bool (true or false)
- List: Sequence of values with numbering starts with 0. Always denoted within [ ]. Ex. [“test1”, “test2”] where “test1” is value 0 and test2 is numbered as 1.
- Tuple
- Map: Group of values i pairs. Always denoted within {}. Ex. {“name”=”John”, “age”=”43”}
- Object
- Null: Null value. Denotes absence or omission.

Terraform always converts number and bool to strings when these values supplied to string variable. Means 1,2,true,false,50 are all valid string variable values.

If you don’t define the default value of a variable then on `terraform plan` or `apply` it will prompt you to supply a value for that variable.

Outputs file

`outputs.tf`

Tell terraform apply to which values to output
Those will be shown on screen or can queried using terraform output

Terminology

Ideompodent

It means even if the same code is applied multiple times, the result remains the same.Terraform supports it.

Day 0 activities

It means provisioning and configuring the infrastructure i.e. initial build.

Day 1 activities

Its OS or software configuration tasks to be done once infrastructure is provisioned. Like patching etc.

Cloud-agnostic

It means a single config can be used to managed multiple cloud providers or even handle cross-cloud dependencies. Used to build fault tolerant infra.

HCL

HashiCorp configuration language

Commands

terraform init

Run it in the project directory with the configuration files. This will download the correct provider plug-ins for the project.

- Plugin download and install
- Backend initialization
- Modules initialization

Terraform installs providers from the Terraform Registry by default.
With terraform 0.13 `terraform init` can download community providers automatically.

1. `-upgrade`: Upgrade all downloaded plugins to their latest version while honouring the version constraints mentioned in the configuration.

terraform fmt

Automatically updates configurations in the current directory for readability and consistency.
By default scans all files in CWD and formats them.

1. `terraform fmt <direcotry>`: Format files in given dir rather than CWD
2. `-diff`: Display diff of changes
3. `-check`: Only checks if the file is formatted. Non-zero exit status if not.
4. `-recursive`

It is recommended to run `terraform fmt` after you upgrade terraform platform. Upgrades may bring small formatting changes and it's good practice to make sure your code is aligned with it.

terraform validate

Makes sure your configuration is syntactically valid and internally consistent. Anyway these checks are done when you run `terraform plan`

terraform plan

It shows the actions will be performed once you run a `terraform apply`
Use `-out=filename` to save the plan. This file can be supplied to apply the command to execute. Helps in automation cases.

1. Destroy mode: Use `-destroy`
2. Refresh-only: Use `-refresh-only.` Update terraform state and output values of objects changed outside terraform
3. `-target=resource.id`: Allows to perform an operation on a specific resource.
4. `-refresh-false`: Do not refresh

terraform apply

It creates real resources as well as a state file that compares future changes in your configuration files to what actually exists in your deployment environment.
It first checks the existing state if any, sync with the real-world infra, checks config, proposes a changeset.

-/+ means that Terraform will destroy and recreate the resource
+ means create a resource

- means destroy the resource
~ means update-in-place

1. `-auto-approve`: Removes the confirmation prompt
2. `terrafom apply filename`: Also does not prompt and apply changes in plan file. For automation!
3. `-replace resource`: To instruct tf to replace some degraded resources even if there is no config change for them in code. (Its better alternative to `terraform taint` command)
4. `-target=resource.id`: Allows to perform an operation on a specific resource.

terraform output

Shows output values defined in `outputs.tf`
Those are shown on the console as well when you run `apply` command.
Output marked as **sensitive** will be not shown in the console outputs of terraform commands but it will still be visible and saved as plain text in the state file.

terraform show

Inspect the current state using terraform show

terraform state

For advanced state management.

1. `list`: List all/particular resource
2. `show`: Show single resource details
3. `refresh`: sync to real-world infra state
4. `pull`: Read from remote state file.
5. `push`: Update remote state from local state file
6. `mv source dest`: Renaming the resource identified in the state file. Or move from one state file to another state file.
7. `rm resource`: Removing tracking of resource in the state file. It won't delete resources in the real world. Make sure you delete the respective resource code as well or terraform will create resources in the next plan.
8. `replace-provider from_provide_fqn to_provider_fqn`: Change provider for all resources

The remote state is loaded into the local machine's memory only when it is being used. It will not be downloaded and saved on local machine.

terraform destroy

Deletes all resources managed by terraform config.
Its kind of the reverse of terraform apply. It walks the dependency graph and deletes resources in dependency order.

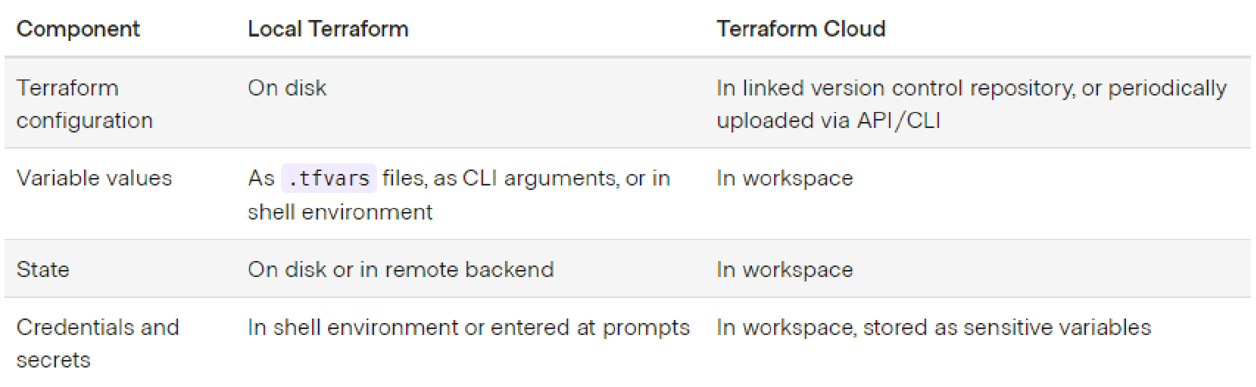
terraform login

log in to your Terraform Cloud account with the Terraform CLI in your terminal. It will open up the browser so that you can log in to terraform cloud account and generate an API token. That API token you need to come back to the terminal and supply it to this command as input.

terraform workspace

Use the same config file in different workspaces to have a different set of physical resources. The use case is spinning identical infra in different environments so that changes can be tested in one workspace before applying to the production workspace.

1. `new kerneltalks`: Creates new workspace and switched to it
2. `select kerneltalks`: Switch to an existing workspace named kerneltalks
3. `list`: shows all workspaces and * against the current one
4. `delete`: Delete workspace. You can not delete the active workspace. Always switch to another workspace and then delete
5. `show`: Shows name of the current workspace.



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

Snip from https://www.terraform.io/docs/cloud/workspaces/index.html
Workspaces are enabled with remote operations by default in terraform cloud. Terraform runs 9plan and apply) happens on tf's own disposable servers when using tf cloud

- Understand the concept of run triggers between workspaces.

- State field of workspaces are stored inside `terraform.tfstate.d` directory. You will see directories with the same name as workspace inside it.
- Due to which local workspaces does not provide strong separation between states. Hence remote workspaces are recommended.
- You can not delete the default workspace
- When you create a new workspace, you will automatically switch to it.
- You can not delete workspace while in it. Switch to another workspace and then delete the former one.

terraform import

Import existing resources into terraform management. It will identify the exiting resource by ID and add them to the state file with the provided logical name.
It does not add the configuration in the state file, that needs to be done manually.

```
1. import aws_instance.server1 i-asde3521
```

terraform taint

Tell terraform to mark resource for recreation. Use case - when you know a particular resource is in bad shape (maybe due to some configuration) terraform will not know about it. Terraform believes resource is good if it matching all the details in the state file. So if you know its ad you need to taint them so that terraform will re-create them in the next apply.

```
1. taint <address>
2. untaint <address>
```

Use untaint to remove tainting applied by mistake.

There are a couple of ways resource can get tained by terraform apart from above command -

1. Resource creation is not finished within the default timeout. The resource may get created by tf will mark it as tained. Increase timeout in such cases.
2. Tf crashes during apply command. Very rare case.

terraform console

Provides interactive console to evaluate expressions.

```
# terraform console
> 1+5
6
> exit
```

terraform graph

generate a visual representation of either a configuration or execution plan.
Output in DOT format. Copy it to graphviz.org to get visual graphs.

terraform providers

View specified provider versions constraints in current configurations.

terraform get

Download and/or update the modules.

Global options

`-chdir`: Override the CWD and work in the directory mentioned in the argument.
`-help`
`-version`
`TF_LOG` (Env variable)
    Values can be TRACE, DEBUG, INFO, WARN and ERROR.
`TF_LOG_PATH` (Env variable)
    Desired file location to save debug logs. Default it sents to STDERR
`crash.log`
    Terraform creates these debug logs with a panic report when it crashes.

Terraform Cloud

Hosted at https://app.terraform.io
Free - small teams
Paid - Mid-sized businesses with additional features

Provides -
1. centralized storage for state files and input variables.
2. Private registry for modules
3. Approving changes to infra
4. Access control for tf configs.

It needs to be defined in terraform block as a remote backend.

```
terraform {
  backend "remote" {
    organization = "my-org"
    workspaces {
      prefix = "my-app-"
    }
  }
}
```

Then switch workspace and plan to see if its good to go

```
$ terraform workspace select my-app-dev
Switched to workspace "my-app-dev".

$ terraform plan
Running plan remotely in Terraform Enterprise.

Output will stream here. To view this plan in a browser, visit:
https://app.terraform.io/my-org/my-app-dev/.../
Refreshing Terraform state in-memory prior to plan...
# ...
Plan: 1 to add, 0 to change, 0 to destroy.
```

Then raise PR. In PR, terraform cloud automatically adds checks and indicates if the check passed and there is no conflict. It makes it easy to decide if PR to be approved and merged or not.
Lastly, when PR is merged, you can `apply it` in terraform cloud console and see only outputs.

Terraform enterprise

For large enterprises.
Its self hosted private instance of terraform cloud application with no resource limit, SAML SSO and audit logging.

Terraform pricing plans comparison:
https://www.datocms-assets.com/2885/1602500234-terraform-full-feature-pricing-tablev2-1.pdf

Sentinel

Policy as a code framework by terraform—paid feature.
Policies are written in sentinel language.
Policies run after `plan` but before it can be confirmed or before `apply`.

Assorted

The logic behind terraform build

Terraform builds a graph of all resources, which helps it to determine the dependencies of resources. While `apply` or `destroy` if walks the graph and accordingly create or delete resources. It also uses parallelism to limit concurrent operations as it walks the graph for a performance boost. Default is ten operations.

Terraform built-in functions

Refer https://www.terraform.io/docs/language/functions/index.html
Go through all functions at least once.
Terraform does not support user-defined custom build functions. It only supports in-built functions.

Backends

It's not a mandatory configuration to define.

1. Standard backend.
    a. Local backends
    b. Its default backend terraform assumes when no backend explicitly defined in code.
2. Enhanced backends.
    a. Backends offering standard functionalities along with remote operations
    b. S3 backend supports state locking with the use of DynamoDB

- When changing the backend type of the existing terraform initialised directory, you should get an option to move your config/files to the new backend.
- Backend can be configured with **partial configuration,** and the rest of the configuration can be supplied on the run.
- Github is not a supported backend type

Backends that supports state locking:

1. Azure rm
2. Hashicorp consul
3. S3 via DDB
4. Terraform enterprise
5. Postgresql
6. GCS
7. HTTP endpoints
8. Etcdv3
9. Tencent COS
10. Kubernetes secrets with ease resource

Backends that does not support state locking:

1. Etcd
2. artifactory

Modules

All module blocks need a **source** defined.'
Modules support below backends :

1. Local path
2. TF registry (Supports module versioning)
3. Github
4. S3
5. Bitbucket
6. GCS
7. HTTP URLs

- Module version is not a mandatory constraint.
- If you have modules downloaded in the directory then repetitive terraform init will not download the modules.
- Module output can be referred to as: `module.module_name.output.value`

The module should fulfil the below criteria to be part of the registry

1. Should be on Git
2. Should follow naming convention: `terraform-<provider>-<name>`
3. Should have standard module structure
4. Should follow verion name X.Y.Z.Can have v in front.