Contents

[Windows Installation: 1](#_Toc42160223)

[Commands 1](#_Toc42160224)

[Examples 2](#_Toc42160225)

[Terraform File Organization 2](#_Toc42160226)

[Setup Terraform DEV Environment 5](#_Toc42160227)

# Windows Installation:

* choco install terraform (from powershell 🡪 Run as Administrator)
* choco install terraform --version=0.12.5
* choco install graphviz -- This is to generate a Resource Graph

# Commands

* terraform init (required whenever there is a change in provider)
* terraform validate .
* terraform plan -- generates the Plan
* terraform plan -out xyz.plan -- It generates the Plan and saves into a file
* terraform apply -- It executes the Template and creates the environment
* terraform destroy – Completely destroys the environment
* terraform plan -var “accessKey=sds” -var “secretkey=hhh”
* terraform plan -var “accessKey=sds” -var “secretkey=hhh” -var-file=uat/uat.tfvars
* terraform plan -var “accessKey=sds” -var “secretkey=hhh” -out xyz.plan .
* terraform apply xyz.plan
* terraform graph | dot -Tpng > graph.png -- To generate the graph

# Examples

1. Create a folder
2. Create .tf file
3. cd to the folder
4. Execute Terraform commands

**Introduction**

* Terraform is a tool for building, changing, and versioning infrastructure safely and efficiently.
* Terraform can manage existing and popular service providers as well as custom in-house solutions.
* Configuration files describe to Terraform the components needed to run a single application or your entire datacenter.
* Terraform generates an execution plan describing what it will do to reach the desired state, and then executes it to build the described infrastructure.
* As the configuration changes, Terraform is able to determine what changed and create incremental execution plans which can be applied.
* 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.
* **Why Terraform?**
* **Terraform is an "orchestration" tool, not an "automation tool:"** Automation is a task completed without human intervention, while Orchestration means, taking a task and creating a workflow, or running several automated tasks called as processes.
* For example, orchestration is a way of combining multiple automation tasks to create IP or creating a security group
* **Terraform is "Declarative" not "Procedural/Imperative:"** These are contrasting programming patterns.
* Declarative programming does not control the flow of the program, you just say what you want and not say how to do it.
* Procedural programming, on the other hand, you define the whole process and provide the steps how to do it.
* **Terraform follows Client Only Architecture, not Client/Server Architecture**: Chef, Ansible all follow client/server architecture. You issue commands on a client system and it executes your commands and stores the state of your system.
* The server talks to agents, which must installed on every instance you want to configure. With this architecture, moving parts coming as a perk and may cause new failure modes. While Terraform uses the cloud provider API's to configure your infrastructure.
* **Terraform has Multi-Provider Support**: Terraform provides convenience to switch between different cloud providers like Google, AWS, Open Stack, Azure. Terraform allows you to write code specific to each provider.

**Key Features**

* **Infrastructure as Code**
  + Infrastructure is described using a high-level configuration syntax. This 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.
* **Execution Plans**
  + Terraform has a "planning" step where it generates an execution plan. The execution plan shows what Terraform will do when you call apply. This lets you avoid any surprises when Terraform manipulates infrastructure.
* **Resource Graph**
  + 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.

**Terraform Building Blocks**

* **Providers**

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

* **Resources -** Resource is part of the infrastructure create on the provider. Every Provider provides resources.
* **Outputs**
* **Variables -** To give options for the user to enter different values to resources.
* **Datasources**
* Terraform templates are written in Custom DSL. This DSL mostly looks like JSON.
* Terraform templates end with .tf extension

**Resource Dependencies**

* When Resource A requires resource B to be present (or already existing) this is called as dependency
* In Terraform terms you have to create resource B before resource A
* To Demonstrate this lets add subnets to VPC. To create subnet resource vpc id is required (vpc has to be existing)
* To create resource dependencies use the following expression

"${<resource-type>.<resource-name>.<attribute-name>}" "${aws\_vpc.myvpc.id}"

**Standard Module Structure**

* main.tf 🡪 providers and Resources
* variables.tf 🡪 All the Variables information
* outputs.tf 🡪 Outputs to be displayed

Normally we have above files when we develop Terraform templates, We maintain the same structure when we develop modules also

**Publish to Terraform Registry**

Requirements

* **Standard module structure.** The module must adhere to the standard module structure. This allows the registry to inspect your module and generate documentation, track resource usage, parse submodules and examples, and more.
* **GitHub**. The module must be on GitHub and must be a public repo. This is only a requirement for the public registry.
* **Named terraform-<PROVIDER>-<NAME>**. Module repositories must use this three-part name format, where <NAME> reflects the type of infrastructure the module manages and <PROVIDER> is the main provider where it creates that infrastructure. The <NAME> segment can contain additional hyphens. Examples: terraform-google-vault or terraform-aws-ec2-instance.
* **x.y.z tags for releases.** The registry uses tags to identify module versions. Release tag names must be a semantic version, which can optionally be prefixed with a v. For example, v1.0.4 and 0.9.2. To publish a module initially, at least one release tag must be present.

Semantic Versioning major.minor.patch

* [Terraform Registry](https://registry.terraform.io/)
  + Login using github credentials
  + Publish module

