**Practical No. 05**

**Practical Name :** Understand terraform lifecycle, core concepts/terminologies and install it on a Linux Machine.

**Aim**: To understand terraform lifecycle, core concepts/terminologies and install it on a Linux Machine.

# Theory :

* **Terraform :**
* Terraform is our tool of choice to manage the entire lifecycle of infrastructure using [infrastructure as code.](https://www.fairwinds.com/blog/why-infrastructure-as-code-kubernetes) That means declaring infrastructure components in configuration files that are then used by Terraform to provision, adjust and tear down infrastructure in various cloud providers.
* Terraform is HashiCorp’s infrastructure as code tool. It lets you define resources and infrastructure in human-readable, declarative configuration files, and manages your infrastructure’s lifecycle. Using Terraform has several advantages over manually managing your infrastructure:
  + Terraform can manage infrastructure on multiple cloud platforms.
  + The human-readable configuration language helps you write infrastructure code quickly.
  + Terraform's state allows you to track resource changes throughout your deployments.
  + You can commit your configurations to version control to safely collaborate on infrastructure.

# Terraform Lifecycle :

Lifecycle is a nested block that can appear within a resource block. The lifecycle block and its contents are meta-arguments, available for all resource blocks regardless of type.

The following arguments can be used within a lifecycle block:

# [create\_before\_destroy](https://www.terraform.io/docs/language/meta-arguments/lifecycle.html#create_before_destroy) (bool) –

* By default, when Terraform must change a resource argument that cannot be updated in- place due to remote API limitations, Terraform will instead destroy the existing object and then create a new replacement object with the new configured arguments.
* The create\_before\_destroy meta-argument changes this behavior so that the new replacement object is created first, and the prior object is destroyed after the replacement is created.
* This is an opt-in behavior because many remote object types have unique name requirements or other constraints that must be accommodated for both a new and an old

object to exist concurrently. Some resource types offer special options to append a random suffix onto each object name to avoid collisions, for example. Terraform CLI cannot automatically activate such features, so you must understand the constraints for each resource type before using create\_before\_destroy with it.

# [prevent\_destroy](https://www.terraform.io/docs/language/meta-arguments/lifecycle.html#prevent_destroy) (bool) –

* This meta-argument, when set to true, will cause Terraform to reject with an error any plan that would destroy the infrastructure object associated with the resource, as long as the argument remains present in the configuration.
* This can be used as a measure of safety against the accidental replacement of objects that may be costly to reproduce, such as database instances. However, it will make certain configuration changes impossible to apply, and will prevent the use of the terraform destroy command once such objects are created, and so this option should be used sparingly.
* Since this argument must be present in configuration for the protection to apply, note that this setting does not prevent the remote object from being destroyed if the resource block were removed from configuration entirely: in that case, the prevent\_destroy setting is removed along with it, and so Terraform will allow the destroy operation to succeed.

# [ignore\_changes](https://www.terraform.io/docs/language/meta-arguments/lifecycle.html#ignore_changes) (list of attribute names) –

* By default, Terraform detects any difference in the current settings of a real infrastructure object and plans to update the remote object to match configuration.
* The ignore\_changes feature is intended to be used when a resource is created with references to data that may change in the future, but should not affect said resource after its creation. In some rare cases, settings of a remote object are modified by processes outside of Terraform, which Terraform would then attempt to "fix" on the next run. In order to make Terraform share management responsibilities of a single object with a separate process, the ignore\_changes meta-argument specifies resource attributes that Terraform should ignore when planning updates to the associated remote object.
* The arguments corresponding to the given attribute names are considered when planning a create operation, but are ignored when planning an update. The arguments are the relative address of the attributes in the resource. Map and list elements can be referenced using index notation, like tags["Name"] and list[0] respectively.

# Basic CLI Features :

The command line interface to Terraform is via the terraform command, which accepts a variety of subcommands such as terraform init or terraform plan. A full list of all of the supported subcommands is in the navigation section of this page.

We refer to the terraform command line tool as "Terraform CLI" elsewhere in the documentation. This terminology is often used to distinguish it from other components you might use in the Terraform product family, such as Terraform Cloud or the various Terraform providers, which are developed and released separately from Terraform CLI.

To view a list of the commands available in your current Terraform version, run terraform with no additional arguments:

* Main Commands :

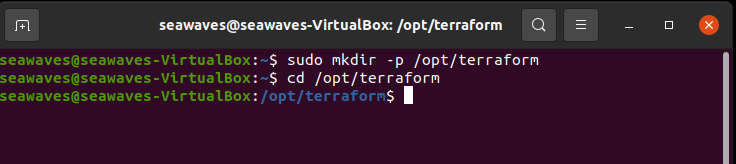
|  |  |
| --- | --- |
| init | Prepare your working directory for other commands |
| validate | Check whether the configuration is valid |
| plan | Show changes required by the current configuration |
| apply | Create or update infrastructure |
| destroy | Destroy previously-created infrastructure |

* All other Commands :

|  |  |
| --- | --- |
| console | Try Terraform expressions at an interactive command prompt |
| fmt | Reformat your configuration in the standard style |
| force-unlock | Release a stuck lock on the current workspace |
| get | Install or upgrade remote Terraform modules |
| graph | Generate a Graphviz graph of the steps in an operation |
| import | Associate existing infrastructure with a Terraform resource |
| login | Obtain and save credentials for a remote host |
| logout | Remove locally-stored credentials for a remote host |
| output | Show output values from your root module |
| providers | Show the providers required for this configuration |
| refresh | Update the state to match remote systems |
| show | Show the current state or a saved plan |
| state | Advanced state management |
| taint | Mark a resource instance as not fully functional |
| test | Experimental support for module integration testing |
| untaint | Remove the 'tainted' state from a resource instance |
| version | Show the current Terraform version |
| workspace | Workspace management |

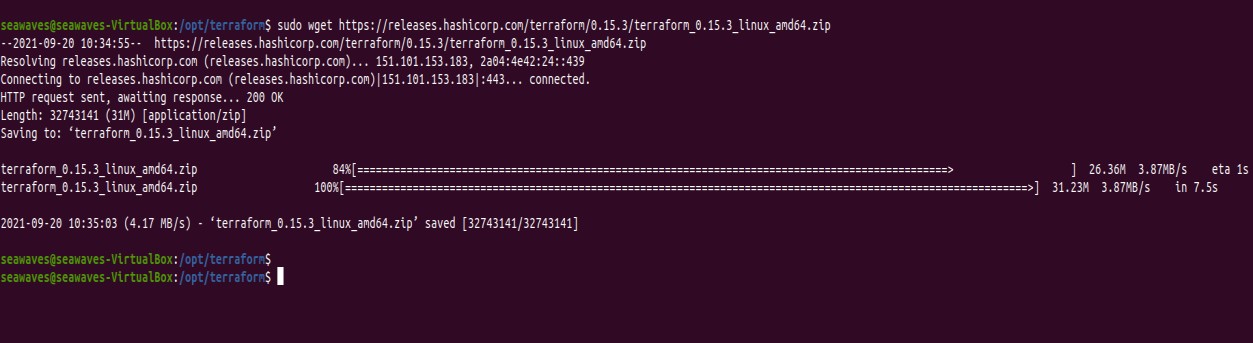
**Steps to install terraform on linux machine :**

1. Create a working directory sudo mkdir -p /opt/terraform cd /opt/terraform

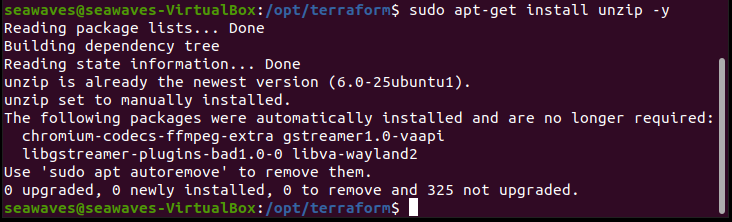


1. Download Terraform from Hasicorp website

sudo wget https://releases.hashicorp.com/terraform/0.15.3/terraform\_0.15.3\_linux\_amd64.zip

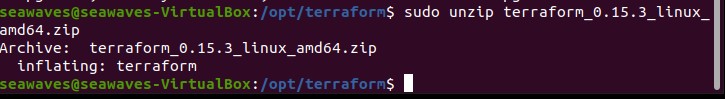


1. Install unzip utility sudo apt-get install unzip -y



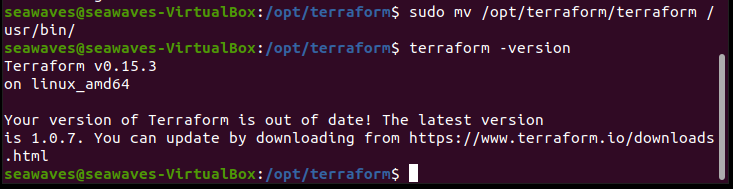
1. Unzip Terraform Zip file

sudo unzip terraform\_0.15.3\_linux\_amd64.zip



1. Add terraform to PATH

sudo mv /opt/terraform/terraform /usr/bin/ terraform -version



# Conclusion:

Successfully, Understood the terraform lifecycle and installed it on Linux machine.