Day 61- Terraform

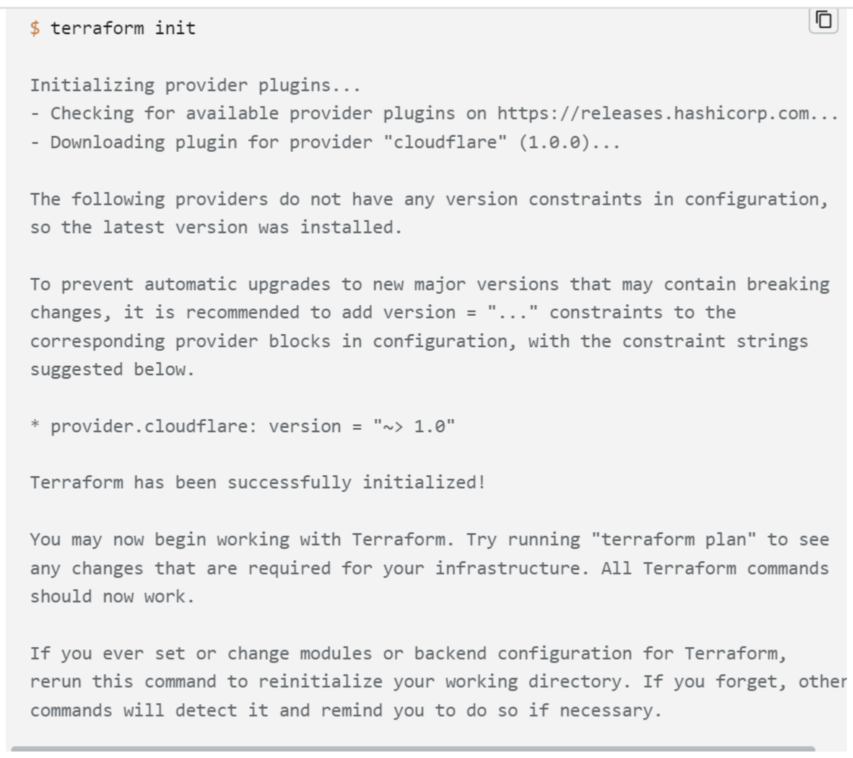
Hope you've already got the gist of What Working with Terraform would be like. Let's begin with day 2 of Terraform!

Task 1:

find purpose of basic Terraform commands which you'll use often

1. terraform init:-

The terraform init command initializes a working directory containing Terraform configuration files. This is the first command that should be run after writing a new Terraform configuration or cloning an existing one from version control. It is safe to run this command multiple times.



```
$ terraform init

Initializing provider plugins...
- Checking for available provider plugins on https://releases.hashicorp.com...
- Downloading plugin for provider "cloudflare" (1.0.0)...

The following providers do not have any version constraints in configuration,
so the latest version was installed.

To prevent automatic upgrades to new major versions that may contain breaking
changes, it is recommended to add version = "..." constraints to the
corresponding provider blocks in configuration, with the constraint strings
suggested below.

* provider.cloudflare: version = "~> 1.0"

Terraform has been successfully initialized!

You may now begin working with Terraform. Try running "terraform plan" to see
any changes that are required for your infrastructure. All Terraform commands
should now work.

If you ever set or change modules or backend configuration for Terraform,
rerun this command to reinitialize your working directory. If you forget, other
commands will detect it and remind you to do so if necessary.
```

2. terraform init –upgrade: -

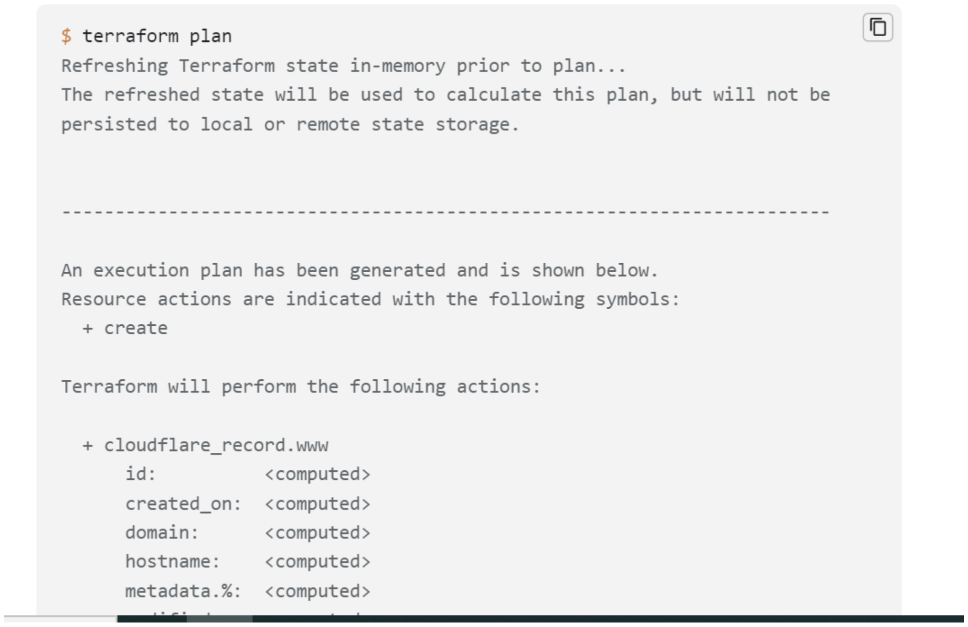
During init, terraform searches the configuration for both direct and indirect references to providers and attempts to install the plugins for those providers. After successful installation, terraform writes information about the selected providers to the dependency lock file. You should commit this file to your version control system to ensure that when you run terraform init again in future Terraform will select exactly the same provider versions.

Use the **-upgrade** option if you want Terraform to ignore the dependency lock file and consider installing newer versions.

3. terraform plan :-

The terraform plan command creates an execution plan, which lets you preview the changes that Terraform plans to make to your infrastructure. By default, when terraform creates a plan it:

- 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 plan
Refreshing Terraform state in-memory prior to plan...
The refreshed state will be used to calculate this plan, but will not be
persisted to local or remote state storage.


------------------------------------------------------------------------

An execution plan has been generated and is shown below.
Resource actions are indicated with the following symbols:
  + create

Terraform will perform the following actions:

  + cloudflare_record.www
      id:          <computed>
      created_on:  <computed>
      domain:      <computed>
      hostname:    <computed>
      metadata.%:  <computed>
```

The plan command alone does not actually carry out the proposed changes. You can use this command to check whether the proposed changes match what you expected before you apply the changes or share your changes with your team for broader review.

If terraform detects that no changes are needed to resource instances or to root module output values, terraform plan will report that no actions need to be taken.

4. terraform apply:

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

When you run terraform apply without passing a saved plan file, terraform automatically creates a new execution plan as if you had run terraform plan, prompts you to approve that plan, and takes the indicated actions. You can use all of the planning modes and planning options to customize how Terraform will create the plan.

You can pass the **-auto-approve** option to instruct Terraform to apply the plan without asking for confirmation.

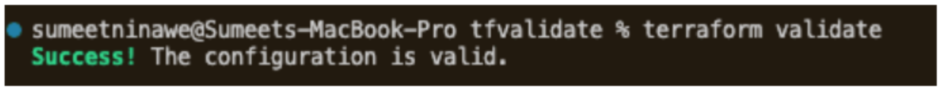
When you pass a saved plan file to terraform apply, terraform takes the actions in the saved plan without prompting you for confirmation.

5. **terraform validate :**

The validate command, on the other hand, is used to validate the configuration internally i.e., locally on the host system. Its focus is on validating the Terraform configuration files for syntax and internal consistencies.

Thus, validate command does not depend on any state file or information regarding deployed command.

When we run terraform validate in the Terraform root directory, it simply outputs if the configuration is valid or not. For a valid configuration, the output is shown below.

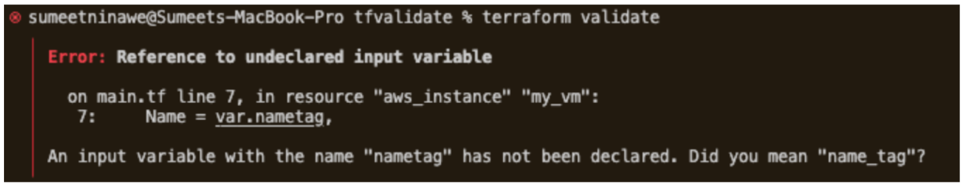


```
sumeetninawe@Sumeets-MacBook-Pro tfvalidate % terraform validate
Success! The configuration is valid.
```

However, if the configuration is invalid, it provides us with the details as shown in the below screenshot.

The details include:

6. File name
7. Line of code that causes the error
8. Summary of the error message
9. Details



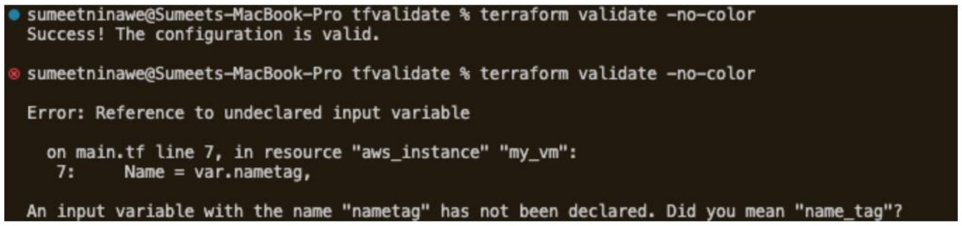
```
sumeetninawe@Sumeets-MacBook-Pro tfvalidate % terraform validate

  Error: Reference to undeclared input variable

    on main.tf line 7, in resource "aws_instance" "my_vm":
     7:     Name = var.nametag,

  An input variable with the name "nametag" has not been declared. Did you mean "name_tag"?
```

The Terraform validate command comes with a couple of options. They are as follows:

1] **-json** – outputs the error details in a JSON format. The output produced in JSON format is used as an input to another program which may trigger appropriate automation workflows or any associated program.

2] **-no-color** – produces the output without any color.

Let us use the -no-color flag and observe the output in success and error cases. If you compare the output below to the ones above, it does not contain any word which is highlighted using colors. For example, “Success!” is not Green, and “Error:” is not Red.



```
● sumeetninawe@Sumeets-MacBook-Pro tfvalidate % terraform validate -no-color
  Success! The configuration is valid.

⊗ sumeetninawe@Sumeets-MacBook-Pro tfvalidate % terraform validate -no-color

  Error: Reference to undeclared input variable

    on main.tf line 7, in resource "aws_instance" "my_vm":
     7:     Name = var.nametag,

  An input variable with the name "nametag" has not been declared. Did you mean "name_tag"?
```

10.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 the current directory and apply formatting to all .tf files.

11.terraform destroy

The terraform destroy command is used to destroy the Terraform-managed infrastructure.

Destroys all the infrastructure created by Terraform, freeing up resources and ensuring that you're not paying for unused infrastructure. This command should be used with caution as it will delete all the resources described in the configuration file

What are the main competitors of Terraform?

AWS CloudFormation: AWS CloudFormation is an IaC tool provided by Amazon Web Services (AWS). It allows users to define and manage infrastructure resources in a declarative way using JSON or YAML templates. CloudFormation supports a wide range of AWS services and is tightly integrated with other AWS tools and services.

Ansible: Ansible is an open-source IaC tool that uses YAML scripts to define and manage infrastructure resources. Ansible can be used for provisioning, configuration management, and application deployment across multiple platforms and cloud providers.

Kubernetes: Kubernetes is another IT automation alternative to Terraform. It is an open-source solution that enables automated deployment, management, and scaling of containerized applications. It simplifies container communication. It facilitates container discovery and management within an application.

Puppet: Puppet is another open-source IaC tool that uses a declarative language to define and manage infrastructure resources. Puppet has a large community of users and contributors and supports a wide range of platforms and cloud providers.

Chef: Chef is an open-source IaC tool that uses Ruby scripts to define and manage infrastructure resources. Chef is particularly well-suited for managing large-scale infrastructures and supports a wide range of platforms and cloud providers.

Packer: Packer creates identical machine images for multiple platforms from a single source configuration. A common use case is creating golden images for organizations to use in cloud infrastructure.

Cloud Foundry: Cloud Foundry is an open-source cloud application platform that makes it faster and easier to build, test, deploy, and scale apps in your choice of cloud, framework, and language. Remove the cost and complexity associated with configuring, managing, and securing infrastructure for your app with Cloud Foundry.