Terraform CLI Cheat Sheet

About Terraform CLI

Terraform, a tool created by Hashicorp in 2014, written in Go, aims to build, change and version control your infrastructure. This tool have a powerfull and very intuitive Command Line Interface.

Installation

Install through curl

```
$ curl -O https://releases.hashicorp.com/terraform/
0.11.10/terraform_0.11.10_linux_amd64.zip
$ sudo unzip terraform_0.11.10_linux_amd64.zip
 -d /usr/local/bin/
$ rm terraform_0.11.10_linux_amd64.zip
```

OR install through tfenv: a Terraform version manager

First of all, download the tfenv binary and put it in your PATH.

```
$ git clone https://github.com/Zordrak/tfenv.git
 ~/.tfenv
$ echo 'export PATH="$HOME/.tfenv/bin:$PATH"'
 >> $HOME/bashrc
```

Then, you can install desired version of terraform:

```
$ tfenv install 0.11.10
```

Usage

Show version

```
$ terraform --version
 Terraform v0.11.10
```

Init Terraform

```
$ terraform init
```

It's the first command you need to execute. Unless, terraform plan, apply, destroy and import will not work. The command terraform init will install :

- terraform modules
- eventually a backend
- and provider(s) plugins

Init Terraform and don't ask any input

```
$ terraform init -input=false
```

Change backend configuration during the init

```
$ terraform init -backend-config=cfg/s3.dev.tf -
reconfigure
```

`-reconfigure` is used in order to tell terraform to not copy the existing state to the new remote state location.

Get

This command is useful when you have defined some modules. Modules are vendored so when you edit them, you need to get again modules content.

```
$ terraform get -update=true
```

When you use modules, the first thing you'll have to do is to do a terraform get. This pulls modules into the .terraform directory. Once you do that, unless you do another `terraform get -update=true`, you've essentially vendored those modules.

Plan

The plan step check configuration to execute and write a plan to apply to target infrastructure provider.

```
$ terraform plan -out plan.out
```

It's an important feature of Terraform that allows a user to see which actions Terraform will perform prior to making any changes, increasing confidence that a change will have the desired effect once applied.

When you execute terraform plan command, terraform will scan all *.tf files in your directory and create the plan.

Apply

Now you have the desired state so you can execute the plan.

```
$ terraform apply plan.out
```

Good to know: Since terraform v0.11+, in an interactive mode (non CI/CD/autonomous pipeline), you can just execute `terraform apply` command which will print out which actions TF will perform.

By generating the plan and applying it in the same command, Terraform can guarantee that the execution plan won't change, without needing to write it to disk. This reduces the risk of potentially-sensitive data being left behind, or accidentally checked into version control.

```
$ terraform apply
```

Apply and auto approve

```
$ terraform apply -auto-approve
```

Apply and define new variables value

```
$ terraform apply -auto-approve
-var tags-repository_url=${GIT_URL}
```

Apply only one module

```
$ terraform apply -target=module.s3
```

This *-target* option works with *terraform plan* too.

Destroy

```
$ terraform destroy
```

Delete all the resources!

A deletion plan can be created before:

```
$ terraform plan –destroy
```

`-target` option allow to destroy only one resource, for example a S3 bucket :

```
$ terraform destroy -target aws_s3_bucket.my_bucket
```

Debug

The Terraform console command is useful for testing interpolations before using them in configurations. Terraform console will read configured state even if it is remote.

```
$ echo "aws_iam_user.notif.arn" | terraform console
arn:aws:iam::123456789:user/notif
```

Graph

```
$ terraform graph | dot –Tpng > graph.png
```

Visual dependency graph of terraform resources.

Validate

Validate command is used to validate/check the syntax of the Terraform files. A syntax check is done on all the terraform files in the directory, and will display an error if any of the files doesn't validate. The syntax check does not cover every syntax common issues.

```
$ terraform validate
```

Providers

You can use a lot of providers/plugins in your terraform definition resources, so it can be useful to have a tree of providers used by modules in your project.

```
$ terraform providers
.
├── provider.aws ~> 1.24.0
├── module.my_module
│   ├── provider.aws (inherited)
│   ├── provider.null
│   └── provider.template
└── module.elastic
    └── provider.aws (inherited)
```

State

Pull remote state in a local copy

```
$ terraform state pull > terraform.tfstate
```

Push state in remote backend storage

```
$ terraform state push
```

This command is usefull if for example you riginally use a local tf state and then you define a backend storage, in S3 or Consul…

How to tell to Terraform you moved a ressource in a module?

If you moved an existing resource in a module, you need to update the state:

```
$ terraform state mv aws_iam_role.role1 module.mymodul
```

How to import existing resource in Terraform?

If you have an existing resource in your infrastructure provider, you can import it in your Terraform state:

```
$ terraform import aws_iam_policy.elastic_post
arn:aws:iam::123456789:policy/elastic_post
```

Workspaces

To manage multiple distinct sets of infrastructure resources/environments.

Instead of create a directory for each environment to manage, we need to just create needed workspace and use them:

Create workspace

This command create a new workspace and then select it

```
$ terraform workspace new dev
```

Select a workspace

```
$ terraform workspace select dev
```

List workspaces

```
$ terraform workspace list
  default
* dev
  prelive
```

Show current workspace

```
$ terraform workspace show
dev
```

Tools

jq

jq is a lightweight command-line JSON processor. Combined with terraform output it can be powerful.

Installation

For Linux:

```
$ sudo apt-get install jq
```

or

```
$ yum install jq
```

For OS X:

```
$ brew install jq
```

Usage

For example, we defind outputs in a module and when we execute *terraform apply* outputs are displayed:

```
$ terraform apply
...
Apply complete! Resources: 0 added, 0 changed,
 0 destroyed.

Outputs:

elastic_endpoint = vpc-toto-12fgfd4d5f4ds5fngetwe4.
eu-central-1.es.amazonaws.com
```

We can extract the value that we want in order to use it in a script for example. With jq it's easy:

```
$ terraform output -json
{
    "elastic_endpoint": {
        "sensitive": false,
        "type": "string",
        "value": "vpc-toto-12fgfd4d5f4ds5fngetwe4.
        eu-central-1.es.amazonaws.com"
    }
}

$ terraform output -json | jq '.elastic_endpoint.value
"vpc-toto-12fgfd4d5f4ds5fngetwe4.eu-central-1.
es.amazonaws.com"
```

Terraforming

If you have an existing AWS account for examples with existing components like S3 buckets, SNS, VPC … You can use terraforming tool, a tool written in Ruby, which extract existing AWS resources and convert it to Terraform files!

Installation

`$ sudo apt install ruby` or `$ sudo yum install ruby`

and

```
$ gem install terraforming
```

Usage

Pre-requisites :

Like for Terraform, you need to set AWS credentials

```
$ export AWS_ACCESS_KEY_ID="an_aws_access_key"
$ export AWS_SECRET_ACCESS_KEY="a_aws_secret_key"
$ export AWS_DEFAULT_REGION="eu-central-1"
```

You can also specify credential profile in *~/.aws/credentials_s and with _–profile* option.

```
$ cat ~/.aws/credentials
[aurelie]
aws_access_key_id = xxx
aws_secret_access_key = xxx
aws_default_region = eu-central-1
```

```
$ terraforming s3 --profile aurelie
```

Usage

```
$ terraforming --help
Commands:
terraforming alb # ALB
...
terraforming vgw # VPN Gateway
terraforming vpc # VPC
```

Example:

```
$ terraforming s3 > aws_s3.tf
```

Remarks: As you can see, terraforming can't extract for the moment API gateway resources so you need to write it manually.

Authors :

@aurelievache
Cloud Dev(Ops) at Continental

v1.0.2