

## 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
$ 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 ~/.tf
$ echo 'export PATH="$HOME/.tfenv/bin:$PATH"' >> $HOME
```

Then, you can install desired version of terraform:

```
$ tfenv install 0.11.10
Terraform v0.11.10 is already installed
```

#### Show version

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

### Usage

#### 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.${params.target}.tf -reconfigure
```

### 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=tag1 -var tags=tag2
repository_url=${GIT_URL} -var tags=tag1
commit_id=${GIT_COMMIT}
```

### 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

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

### Graph

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

Visual dependency graph of terraform resources.

### State

#### 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.firehose_delivery_role module.mymodule
```

#### 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
aws_iam_policy.elastic_post: Importing from ID "arn:aws:iam::123456789:policy/elastic_post"
aws_iam_policy.elastic_post: Import complete!
  Imported aws_iam_policy (ID: arn:aws:iam::123456789:policy/elastic_post)
aws_iam_policy.elastic_post: Refreshing state... (ID: arn:aws:iam::123456789:policy/elastic_post)
```

Import successful!

The resources that were imported are shown above. These resources are managed by you, your Terraform state and will henceforth be managed by Terraform.

### 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
```

Tools

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!

Install terraforming

```
$ sudo apt install ruby
$ gem install terraforming
```

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 by –profile option.

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

Pass profile name by –profile option

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

Usage

```
$ terraforming --help
Commands:
terraforming alb # ALB
terraforming asg # AutoScaling Group
terraforming cwa # CloudWatch Alarm
terraforming dbpg # Database Parameter Group
terraforming dbsg # Database Security Group
terraforming dbsn # Database Subnet Group
terraforming ec2 # EC2
terraforming ecc # ElastiCache Cluster
terraforming ecsn # ElastiCache Subnet Group
terraforming efs # EFS File System
terraforming eip # EIP
terraforming elb # ELB
terraforming help [COMMAND] # Describe available commands
terraforming iamg # IAM Group
terraforming iamgm # IAM Group Membership
terraforming iamgp # IAM Group Policy
terraforming iamip # IAM Instance Profile
terraforming iamp # IAM Policy
terraforming iampa # IAM Policy Attachment
terraforming iamr # IAM Role
terraforming iamrp # IAM Role Policy
terraforming iamu # IAM User
terraforming iamup # IAM User Policy
terraforming igw # Internet Gateway
terraforming kmsa # KMS Key Alias
terraforming kmsk # KMS Key
terraforming lc # Launch Configuration
terraforming nacl # Network ACL
terraforming nat # NAT Gateway
terraforming nif # Network Interface
terraforming r53r # Route53 Record
terraforming r53z # Route53 Hosted Zone
terraforming rds # RDS
terraforming rs # Redshift
terraforming rt # Route Table
terraforming rta # Route Table Association
terraforming s3 # S3
terraforming sg # Security Group
terraforming sn # Subnet
terraforming snss # SNS Subscription
terraforming snst # SNS Topic
terraforming sqs # SQS
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.

Known issues

Signature expired: xxxx is now earlier than xxx

If, suddently, you obtain an error message ``Signature expired: xxx is now earlier than xxx", like this:

Error: Error refreshing state: 16 error(s) occurred

```
* aws_api_gateway_rest_api.toto_api: 1 error(s) occurred:
* aws_api_gateway_rest_api.toto_api: aws_api_gateway_rest_api: status code: 403, request id: 04c1518e-e489-11e7-ab12-111111111111
* aws_api_gateway_rest_api.api: 1 error(s) occurred
```

Don’t worry it’s not an issue :

in the AWS account/user/credentials in terraform files

BUT it’s an issue in your local machine date and time!

So the solution is, simply, to update your date and time to the good time ;-).

AWS was not able to validate the provided access credentials

If, suddently, you obtain an error message ``AWS was not able to validate the provided access credentials", like this:

```
* data.aws_vpc.vpc-titi: data.aws_vpc.vpc-titi: Auth failed: status code: 401, request id: 9fbd5beb-e065-4933-ba12-111111111111
```

No worries, it’s the same issue as above: your local/VM machine date and time is not uptodate ;-).

Error configuring the backend ``s3": RequestError: send request failed

Again, you changed nothing but suddently you obtain a strange error message:


Initializing the backend...

Error configuring the backend "s3": RequestError: send request failed: caused by: Post https://sts.amazonaws.com/: Parent request failed

Please update the configuration in your Terraform file. If you'd like to update the configuration interactively, update the values in your configuration, run "terraform init -reconfigure".

It caused in reality by the proxy or a temporary issue between your network connectivity and AWS.

Authors :



**@aurelievache**  
Cloud Dev(Ops) at Continental

v1.0.0