**Run terraform command with var-file**

$ cat config/dev.tfvars

name = "dev-stack"

s3\_terraform\_bucket = "dev-stack-terraform"

tag\_team\_name = "hello-world"

$ terraform plan -var-file=config/dev.tfvars

With var-file, you can easily manage environment (dev/stag/uat/prod) variables.

With var-file, you avoid running terraform with long list of key-value pairs ( -var foo=bar )

**Enable version control on terraform state files bucket**

Always set backend to s3 and enable version control on this bucket.

[s3-backend](https://github.com/ozbillwang/terraform-best-practices/blob/master/s3-backend) to create s3 bucket and dynamodb table to use as terraform backend.

**Manage S3 backend for tfstate files**

Terraform doesn't support [Interpolated variables in terraform backend config](https://github.com/hashicorp/terraform/pull/12067), normally you write a seperate script to define s3 backend bucket name for different environments, but I recommend to hard code it directly as below. This way is called as [partial configuration](https://www.terraform.io/docs/backends/config.html#partial-configuration)

Add below code in terraform configuration files.

$ cat main.tf

terraform {

backend "s3" {

encrypt = true

}

}

Define backend variables for particular environment

$ cat config/backend-dev.conf

bucket = "<account\_id>-terraform-states"

key = "development/service-name.tfstate"

encrypt = true

region = "ap-southeast-2"

dynamodb\_table = "terraform-lock"

**Notes on S3**

* bucket - s3 bucket name, has to be globally unique.
* key - Set some meaningful names for different services and applications, such as vpc.tfstate, application\_name.tfstate, etc
* dynamodb\_table - optional when you want to enable [State Locking](https://www.terraform.io/docs/state/locking.html)

After you set config/backend-dev.conf and config/dev.tfvars properly (for each environment). You can easily run terraform as below:

env=dev

terraform get -update=true

terraform init -backend-config=config/backend-${env}.conf

terraform plan -var-file=config/${env}.tfvars

terraform apply -var-file=config/${env}.tfvars

**Manage multiple Terraform modules and environments easily with Terragrunt**

Terragrunt is a thin wrapper for Terraform that provides extra tools for working with multiple Terraform modules. [https://www.gruntwork.io](https://www.gruntwork.io/)

Sample for reference: <https://github.com/gruntwork-io/terragrunt-infrastructure-live-example>

Its README is too long, if you need a quick start, follow below steps:

# Install terraform and terragrunt

# Make sure you are in right aws account

$ aws s3 ls

# use terragrunt to deploy

$ git clone https://github.com/gruntwork-io/terragrunt-infrastructure-live-example.git

$ cd terragrunt-infrastructure-live-example

# for example, you want to deploy mysql in stage non-prod at region us-east-1

$ cd non-prod/us-east-1/stage/mysql

$ terragrunt plan

# Confirm everything works

$ terragrunt apply

So if you followed the setting in terragrunt properly, you don't need to care about the backend state files and variable file path in different environments, even more, you can run terragrunt plan-all to plan all modules together.

**Retrieve state meta data from a remote backend**

Normally we have several layers to manage terraform resources, such as network, database, application layers. After you create the basic network resources, such as vpc, security group, subnets, nat gateway in vpc stack. Your database layer and applications layer should always refer the resource from vpc layer directly via terraform\_remote\_state data srouce.

Notes: in Terraform v0.12+, you need add extra outputs to reference the attributes, otherwise you will get error message of [Unsupported attribute](https://github.com/hashicorp/terraform/issues/21442)

data "terraform\_remote\_state" "vpc" {

backend = "s3"

config = {

bucket = var.s3\_terraform\_bucket

key = "${var.environment}/vpc.tfstate"

region = var.aws\_region

}

}

# Retrieves the vpc\_id and subnet\_ids directly from remote backend state files.

resource "aws\_xx\_xxxx" "main" {

# ...

subnet\_ids = split(",", data.terraform\_remote\_state.vpc.data\_subnets)

vpc\_id = data.terraform\_remote\_state.vpc.outputs.vpc\_id

}

**Turn on debug when you need do troubleshooting**

TF\_LOG=DEBUG terraform <command>

# or if you run with terragrunt

TF\_LOG=DEBUG terragrunt <command>

**Use shared modules**

Manage terraform resource with shared modules, this will save a lot of coding time. No need re-invent the wheel!

You can start from below links:

* [Terraform module usage](https://www.terraform.io/docs/modules/usage.html)
* [Terraform Module Registry](https://registry.terraform.io/)
* [Terraform aws modules](https://github.com/terraform-aws-modules)

Up to Terraform 0.12, Terraform modules didn't support count parameter.

From Terraform 0.13 on this feature is already available for your pleasure!

**Isolate environment**

Sometimes, developers like to create a security group and share it to all non-prod (dev/qa) environments. Don't do that, create resources with different name for each environment and each resource.

variable "application" {

description = "application name"

default = "<replace\_with\_your\_project\_or\_application\_name, use short name if possible, because some resources have length limit on its name>"

}

variable "environment" {

description = "environment name"

default = "<replace\_with\_environment\_name, such as dev, svt, prod,etc. Use short name if possible, because some resources have length limit on its name>

}

locals {

name\_prefix = "${var.application}-${var.environment}"

}

resource "<any\_resource>" "custom\_resource\_name" {

name = "${local.name\_prefix}-<resource\_name>"

...

}

With that, you will easily define the resource with a meaningful and unique name, and you can build more of the same application stack for different developers without change a lot. For example, you update the environment to dev, staging, uat, prod, etc.

Tips: some aws resource names have length limits, such as less than 24 characters, so when you define variables of application and environment name, use short name.

**Use terraform import to include as many resources you can**

Sometimes developers manually created resources. You need to mark these resource and use terraform import to include them in codes.

[terraform import](https://www.terraform.io/docs/import/usage.html)

**Avoid hard coding the resources**

A sample:

account\_number=“123456789012"

account\_alias="mycompany"

region="us-east-2"

The current aws account id, account alias and current region can be input directly via [data sources](https://www.terraform.io/docs/providers/aws/).

# The attribute `${data.aws\_caller\_identity.current.account\_id}` will be current account number.

data "aws\_caller\_identity" "current" {}

# The attribue `${data.aws\_iam\_account\_alias.current.account\_alias}` will be current account alias

data "aws\_iam\_account\_alias" "current" {}

# The attribute `${data.aws\_region.current.name}` will be current region

data "aws\_region" "current" {}

# Set as [local values](https://www.terraform.io/docs/configuration/locals.html)

locals {

account\_id = data.aws\_caller\_identity.current.account\_id

account\_alias = data.aws\_iam\_account\_alias.current.account\_alias

region = data.aws\_region.current.name

}

**validate and format terraform code**

Always run terraform fmt to format terraform configuration files and make them neat.

I used below code in Travis CI pipeline (you can re-use it in any pipelines) to validate and format check the codes before you can merge it to master branch.

script:

- terraform validate

- terraform fmt -check=true -write=false -diff=true

terraform

One more check [tflint](https://github.com/wata727/tflint) you can add

- find . -type f -name "\*.tf" -exec dirname {} \;|sort -u |while read line; do pushd $line; docker run --rm -v $(pwd):/data -t wata727/tflint; popd; done

**Generate README for each module with input and output variables**

You needn't manually manage USAGE about input variables and outputs. A tool named terraform-docs can do the job for you.

Currently original terraform-docs doesn't support terraform 0.12+, follow this issue: <https://github.com/segmentio/terraform-docs/issues/62> for updating.

Now we have a work around.

# [Terraform >= 0.12]

docker run --rm \

-v $(pwd):/data \

cytopia/terraform-docs \

terraform-docs-012 --sort-inputs-by-required --with-aggregate-type-defaults md . > README.md

For details on how to run terraform-docs, check this repository: <https://github.com/cytopia/docker-terraform-docs>

There is a simple sample for you to start [tf\_aws\_acme](https://github.com/BWITS/tf_aws_acme), the README is generatd by terraform-docs

**Update terraform version**

Hashicorp doesn't have a good qa/build/release process for their software and does not follow semantic versioning rules.

For example, terraform init isn't compatible between 0.9 and 0.8. Now they are going to split providers and use "init" to install providers as plugin in coming version 0.10

So recommend to keep updating to latest terraform version

**Run terraform in docker container**

Terraform releases official docker containers that you can easily control which version you can run.

Recommend to run terraform docker container, when you set your build job in CI/CD pipeline.

TERRAFORM\_IMAGE=hashicorp/terraform:0.12.3

TERRAFORM\_CMD="docker run -ti --rm -w /app -v ${HOME}/.aws:/root/.aws -v ${HOME}/.ssh:/root/.ssh -v `pwd`:/app -w /app ${TERRAFORM\_IMAGE}"

${TERRAFORM\_CMD} init

${TERRAFORM\_CMD} plan

Or with terragrunt

# (1) must mount the local folder to /apps in container.

# (2) must mount the aws credentials and ssh config folder in container.

$ docker run -ti --rm -v $HOME/.aws:/root/.aws -v ${HOME}/.ssh:/root/.ssh -v `pwd`:/apps alpine/terragrunt:0.12.3 bash

# cd to terragrunt configuration directory, if required.

$ terragrunt plan-all

$ terragrunt apply-all

**Run test**

Recommend to add [awspec](https://github.com/k1LoW/awspec) tests through [kitchen](https://kitchen.ci/) and [kitchen-terraform](https://newcontext-oss.github.io/kitchen-terraform/).

**Quick start**

Reference: repo [terraform-aws-modules/terraform-aws-eks](https://github.com/terraform-aws-modules/terraform-aws-eks#testing)

**Run test within docker container**

Reference: [README for terraform awspec container](https://github.com/alpine-docker/bundle-terraform-awspec)

**Minimum AWS permissions necessary for a Terraform run**

There will be no answer for this. But with below iam policy you can easily get started.

{

"Version": "2012-10-17",

"Statement": [

{

"Sid": "AllowSpecifics",

"Action": [

"lambda:\*",

"apigateway:\*",

"ec2:\*",

"rds:\*",

"s3:\*",

"sns:\*",

"states:\*",

"ssm:\*",

"sqs:\*",

"iam:\*",

"elasticloadbalancing:\*",

"autoscaling:\*",

"cloudwatch:\*",

"cloudfront:\*",

"route53:\*",

"ecr:\*",

"logs:\*",

"ecs:\*",

"application-autoscaling:\*",

"logs:\*",

"events:\*",

"elasticache:\*",

"es:\*",

"kms:\*",

"dynamodb:\*"

],

"Effect": "Allow",

"Resource": "\*"

},

{

"Sid": "DenySpecifics",

"Action": [

"iam:\*User\*",

"iam:\*Login\*",

"iam:\*Group\*",

"iam:\*Provider\*",

"aws-portal:\*",

"budgets:\*",

"config:\*",

"directconnect:\*",

"aws-marketplace:\*",

"aws-marketplace-management:\*",

"ec2:\*ReservedInstances\*"

],

"Effect": "Deny",

"Resource": "\*"

}

]

}

Depend on your company or project requirement, you can easily update the resources in Allow session which terraform commands should have, and add deny policies in Deny session if some of permissions are not required.

**Tips to deal with lambda functions**

Headache to save python packages from pip install into source codes and generate lambda zip file manually? Here is full codes with solution.

The folder [lambda](https://github.com/ozbillwang/terraform-best-practices/blob/master/lambda) includes all codes, here is the explanation.

$ tree

.

├── lambda.tf # terraform HCL to deal with lambda

├── pip.sh # script to install python packages with pip.

└── source

├── .gitignore # Ignore all other files

├── main.py # Lambda function, replace with yours

├── requirements.txt # python package list, replace with yours.

└── setup.cfg # Useful for mac users who installed python using Homebrew

Replace main.py and requirements.txt with your applications.

**explanation**

After you run terraform apply, it will:

1. install all pip packages into source folder
2. zip the source folder to source.zip
3. deploy lambda function with source.zip
4. because of source/.gitignore, it will ignore all new installed pip packages in git source codes.

This solution is reference from the comments in [Ability to zip AWS Lambda function on the fly](https://github.com/hashicorp/terraform/issues/8344#issuecomment-345807204))

You should be fine to do the same for lambda functions using nodejs (npm install) or other languages with this tip.

You need have python/pip installed when run terraform commands, if you run in terraform container, make sure you install python/pip in it.

**Usage of variable "self"**

Quote from terraform documents:

Attributes of your own resource

The syntax is self.ATTRIBUTE. For example \${self.private\_ip} will interpolate that resource's private IP address.

Note: The self.ATTRIBUTE syntax is only allowed and valid within provisioners.

**One more use case**

resource "aws\_ecr\_repository" "jenkins" {

name = var.image\_name

provisioner "local-exec" {

command = "./deploy-image.sh ${self.repository\_url} ${var.jenkins\_image\_name}"

}

}

variable "jenkins\_image\_name" {

default = "mycompany/jenkins"

description = "Jenkins image name."

}

You can easily define ecr image url (<account\_id>.dkr.ecr.<aws\_region>.amazonaws.com/<image\_name>) with ${self.repository\_url}

Any attributes in this resource can be self referenced by this way.

Reference: <https://github.com/shuaibiyy/terraform-ecs-jenkins/blob/master/docker/main.tf>

**Use pre-installed Terraform plugins**

There is a way to use pre-installed Terraform plugins instead of downloading them with terraform init, the accepted answer below gives the detail:

[Use pre-installed Terraform plugins instead of downloading them with terraform init](https://stackoverflow.com/questions/50944395/use-pre-installed-terraform-plugins-instead-of-downloading-them-with-terraform-i?rq=1)

**Tips to upgrade to terraform 0.12**

terraform 0.12upgrade

If you have any codes older than 0.12, please go through official documents first,

* [terraform Input Variables](https://www.terraform.io/docs/configuration/variables.html), a lot of new features you have to know.
* [Upgrading to Terraform v0.12](https://www.terraform.io/upgrade-guides/0-12.html)
* [terraform command 0.12upgrade](https://www.terraform.io/docs/commands/0.12upgrade.html)
* [Announcing Terraform 0.12](https://www.hashicorp.com/blog/announcing-terraform-0-12)

Then here are extra tips for you.

* Upgrade to terraform 0.11 first, if you have any.
* Upgrade terraform moudles to 0.12 first, because terraform 0.12 can't work with 0.11 modules.
* Define type for each variable, otherwise you will get weird error messages.

**Tips to upgrade to terraform 0.13**

In fact the command terraform 0.13upgrade in terraform v0.13.3 (the latest version currently) doesn't work to convert older versions less than v0.11

So you have to download terraform 0.12 version to do the upgrade. But from hashicorp terraform website, there is only v0.13.x for downloading now.

Here is a simple way if you can run with docker

# cd to the terraform tf files folder, run below commands

# do the upgrade within terraform 0.12 container

$ docker run -ti --rm -v $(pwd):/apps -w /apps --entrypoint=sh hashicorp/terraform:0.12.29

/apps # terraform init

/apps # terraform 0.12upgrade -yes

/apps # exit

# double check with 0.13upgrade

$ terraform 0.13upgrade -yes

$