



Introduction to Terraform (2d)

Michael Bright / @mjbright Consulting



<https://linkedin.com/in/mjbright>



@mjbright



@mjbright

@mjbright CONSULTING

Outline

- Introduction, Configurations, Terraform Workflow, Installation
- State
- Terragrunt
- Modules
- Provisioners
- Best Practices
- Things to avoid
- Recap

Introduction

Introduction

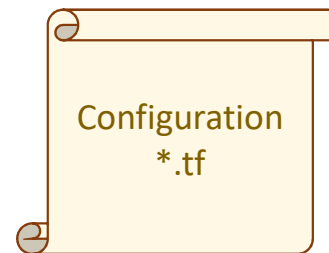


Introduction

Terraform is a tool for provisioning computing infrastructure

It's defining feature is managing “Infrastructure as Code” (IaC)

This means that we define the “desired state” of the infrastructure as code, in configuration files.



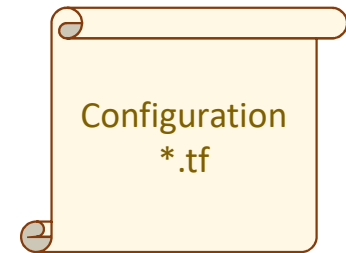
Introduction

“Infrastructure as Code” (IaC) can be achieved through

- Adhoc scripting Shell, Python, Perl, ...
- Config management tools Ansible, Chef, Puppet, Salt, ...
- Server templating tools Docker, Packer, Vagrant
- Orchestration tools Kubernetes, Mesos, Docker Swarm, ECS
- Provisioning tools Terraform, AWS CloudFormation, OpenStack Heat

Introduction

We tell terraform to “apply” the configuration files to create the appropriate resources.



If we tell terraform to “re-apply” the configuration then no further changes will be made – unless the configuration state has drifted.

This means that Terraform is idempotent.

Introduction

“Infrastructure as Code” (IaC) brings

- Automation, repeatability
- Documentation
- Version Control
- Validation, Testability
- Reuse

Introduction

Terraform

- supports many cloud providers, i.e. it is provider agnostic.
- supports many resources for each provider.
- define resources as code in Terraform templates.

Terraform Configurations

Example Terraform configuration

Terraform configurations are written in the Hashicorp Configuration Language (HCL v2) & specify a provider & several resources

The Terraform CLI is a general engine for evaluating & applying configurations, using provider plugins which define & manage a set of resource types.

```
provider 'aws' {  
    region = 'us-west-1'  
}  
  
resource 'aws_instance' 'example' {  
    ami           = 'ami-408c7f28'  
    Instance_type = 't2.micro'  
    Tags = { Name = 'terraform_example' }  
}
```

Example Terraform configuration

The Terraform language is used to declare resources specific to a provider.

```
resource "aws_vpc" "main" {  
    cidr_block = var.base_cidr_block  
}
```

```
<BLOCK TYPE> "<BLOCK LABEL>" "<BLOCK LABEL>" {  
    # Block body: contains arguments or nested blocks  
    <IDENTIFIER> = <EXPRESSION> # Argument  
}
```

<https://www.terraform.io/docs/configuration/index.html>

Example Terraform configuration

Groups of resources can be gathered into modules

A Terraform configuration consists of a *root module*, where evaluation begins, along with a tree of child modules created when one module calls another.

<https://www.terraform.io/docs/configuration/index.html>

Terraform Providers

It is required to have a provider defined for your terraform configuration

We will be using AWS, but Terraform supports many providers

Terraform Providers

AWS

Alibaba Cloud, Azure, DigitalOcean, Exoscale, Google Cloud Platform, Heroku, Oracle Cloud, OVH, Packet, 1&1, Spotinst

Terraform Cloud, Vault, Nomad, Consul

<https://www.terraform.io/docs/providers/index.html>

GitHub, GitLab, Bitbucket,

Kubernetes, Helm, Docker, OpenStack

PostgreSQL, MySQL, MongoDB Atlas

VMware NSX-T, vCloud , vSphere

Chef, Cobbler, Datadog, DNS, HTTP, Local, TLS

@mjbright CONSULTING

Terraform – Provider “AWS” Services

ACM PCA API Gateway Application Autoscaling AppMesh AppSync Athena
Autoscaling Backup Batch Budgets Cloud9 CloudFormation CloudFront
CloudHSM v2 CloudTrail CloudWatch CodeBuild CodeCommit CodeDeploy
CodePipeline Cognito Config Cost & Usage Report Data Lifecycle Manager
(DLM) Database Migration Service (DMS) DataPipeline DataSync Device Farm
Directory Service Direct Connect DynamoDB Accelerator (DAX) DocumentDB
EC2 ECR ECS EFS EKS ElastiCache Elastic Beanstalk **Elastic Load Balancing v2**
(ALB/NLB) Elastic Map Reduce (EMR) ElasticSearch Elastic Transcoder
Firewall Manager (FMS) File System (FSx) Gamelift Glacier Global Accelerator
Glue GuardDuty **IAM** IoT Inspector Kinesis Kinesis Firehose KMS **Lambda**
License Manager Lightsail Macie MQ MediaPackage MediaStore Managed
Streaming for Kafka (MSK) Neptune OpsWorks Organizations Pinpoint Pricing
QuickSight RAM **RDS** Redshift Resource Groups **Route53** Resolver **S3**
Sagemaker Secrets Manager Security Hub SES Service Catalog Service
Discovery Service Quotas Shield SimpleDB SNS SQS SSM Step Function (SFN)
Storage Gateway SWF Transfer **VPC** WAF WAF Regional WorkLink
WorkSpaces XRay

Terraform – Glossary

Argument

Attribute

Backend Block

(Terraform) Configuration (HCL)

Version Data Source Expression Input Variables

Interpolation Module Output Values (Terraform) Provider (Terraform) Registry

Remote Operations

Remote Backend

Repository

Resource

Root Module Root Outputs Speculative Plan State Variables

VCS Workspace

<https://www.terraform.io/docs/glossary.html>

Terraform Installation

Terraform Installation

Terraform is a single static go binary available for macOS, Windows, Linux, downloadable from:

- <https://www.terraform.io/downloads.html>
- <https://github.com/hashicorp/terraform/releases>

On Unix-like OS, simply put it in your PATH, e.g.

```
mv terraform ~/bin/
```

may require:

```
export PATH=$PATH:~/bin
```

<https://github.com/hashicorp/terraform>

Terraform Installation - Version

```
> terraform version  
Terraform v0.12.12  
+ provider.aws v2.33.0  
+ provider.template v2.1.2
```

Your version of Terraform is out of date!

The latest version is 0.12.13.

You can update by downloading from www.terraform.io/downloads.html

Lab 0 – Installing Terraform

In this initial lab we will simply install the Terraform static binary for amd64 architecture and add this to our PATH

Quiz

What does it mean to say that Terraform is idempotent ?

What are some advantages of “*Infrastructure as Code*” ?

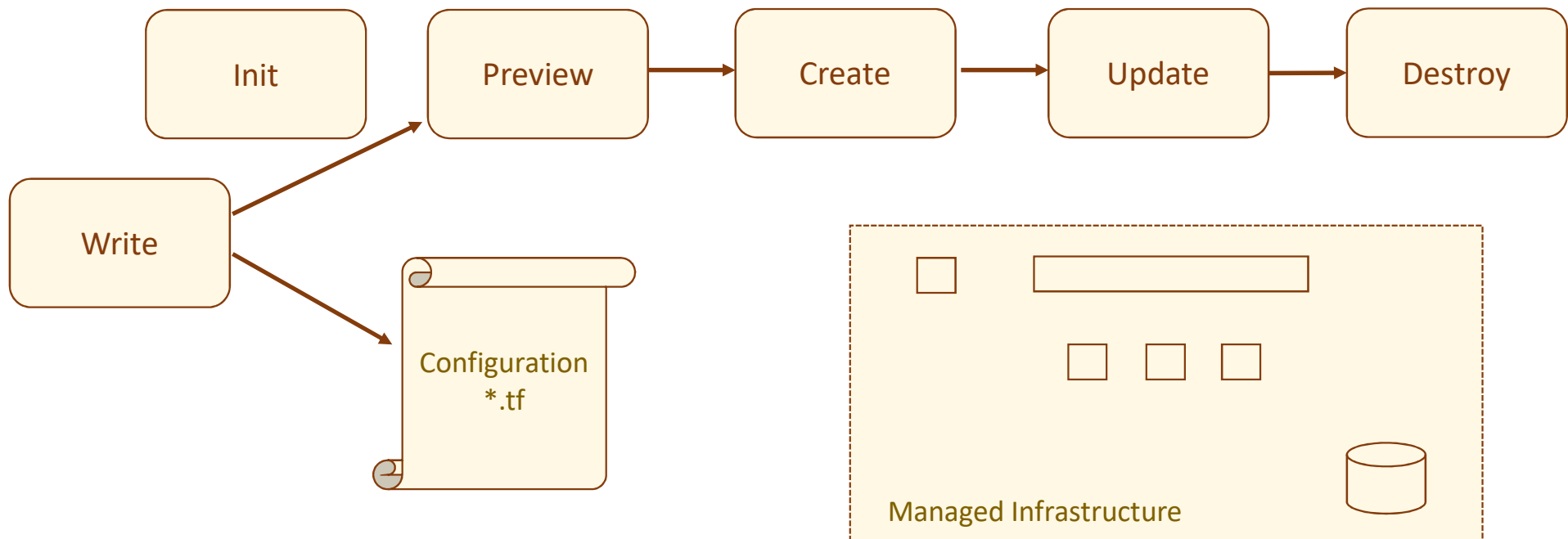
Is Terraform an Open Source project ?

What company is behind Terraform ?

Terraform Workflow

Terraform Workflow

Use of Terraform is according to the workflow

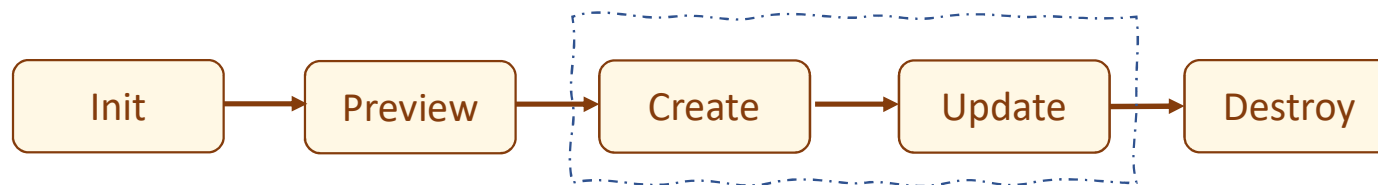


<https://www.terraform.io/docs/commands/index.html>

Terraform Workflow

Use of Terraform is according to the workflow

- terraform init - download and initialize specified providers
- terraform plan - parse and check config
- terraform apply - create or update all resources
- terraform destroy - destroy all resources



<https://www.terraform.io/docs/commands/index.html>

Terraform Workflow

apply	Builds or changes infrastructure	plan	Generate and show an execution plan
console	Interactive console for Terraform interpolations	providers	Prints a tree of the providers used in the config
destroy	Destroy Terraform-managed infrastructure	refresh	Update local state file against real resources
env	Workspace management	show	Inspect Terraform state or plan
fmt	Rewrites config files to canonical format	taint	Manually mark a resource for recreation
get	Download install modules for the config	untaint	Manually unmark a resource as tainted
graph	Create a visual graph of Terraform resources	validate	Validates the Terraform files
import	Import existing infrastructure into Terraform	version	Prints the Terraform version
init	Initialize a Terraform working directory	workspace	Workspace management
output	Read an output from a state file		

Terraform Workflow - Get

The get command is used to download & update modules referenced in the root module.

```
terraform get [-update] [<dir>]
```

-update - reload modules even if already present.

dir - Sets the path of the root module.

Modules download to .terraform unless overwritten by TF_DATA_DIR.

e.g. TF_DATA_DIR=~/.terraform # to avoid version control (can be large)

It is safe to run this command multiple times

<https://www.terraform.io/docs/commands/get.html>

Terraform Workflow - Init

The init command is the first command which should be run to initialize a working directory of configuration files

```
terraform init [<options>] [<dir>]
```

Also downloads modules referenced in root configuration

It is safe to run this command multiple times

<https://www.terraform.io/docs/commands/init.html>

Terraform Workflow - Init

```
> terraform init
Initializing the backend...
Initializing provider plugins...
- Checking for available provider plugins...
- Downloading plugin for provider "aws" (hashicorp/aws) 2.34.0...
```

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.aws: version = "~> 2.34"
```

Terraform has been successfully initialized!

...

<https://www.terraform.io/docs/commands/init.html>

Terraform Workflow - plan

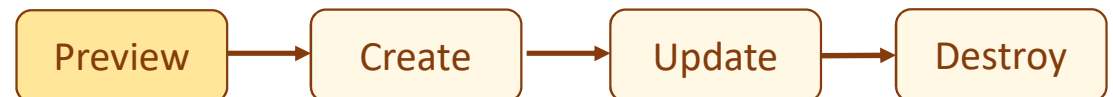
The *plan* command creates an execution plan.

```
terraform plan [-out]
```

Refreshes it's view of the *actual* state, then determines what actions are necessary to achieve the *desired* state

- Checks the configuration file
- Show what will be done when the configuration is applied

Convenient to **preview** the effect of applying a configuration



<https://www.terraform.io/docs/commands/plan.html>

Terraform Workflow - plan

```
> terraform plan
+ aws_instance.example
  ami:                "" => "ami-408c7f28"
  instance_type:      "" => "t2.micro"
  key_name:            "" => "<computed>"
  private_ip:          "" => "<computed>"
  public_ip:           "" => "<computed>"
```

```
Plan:  1 to add, 0 to change, 0 to destroy
```

Terraform Workflow - validate

You can also validate the syntax of files using the “*terraform validate*” command

```
> terraform validate  
Success! The configuration is valid.
```

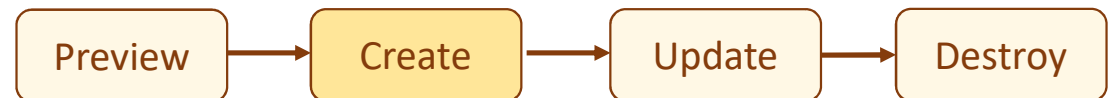
<https://www.terraform.io/docs/commands/plan.html>

Terraform Workflow - apply

The apply command applies the changes required to reach the desired state of the configuration

```
terraform apply [<options>]
```

Creates the required resources, e.g. VM instances



<https://www.terraform.io/docs/commands/apply.html>

@mjbright CONSULTING

Terraform Workflow - apply

```
> terraform apply
+ aws_instance.example: Creating...
  ami:                "" => "ami-408c7f28"
  instance_type:      "" => "t2.micro"
  key_name:           "" => "<computed>"
  private_ip:         "" => "<computed>"
  public_ip:          "" => "<computed>"
aws_instance.example: Creation complete
```

```
Apply complete! Resources: 1 added, 0 changed, 0 destroyed.
```

Terraform Workflow - apply

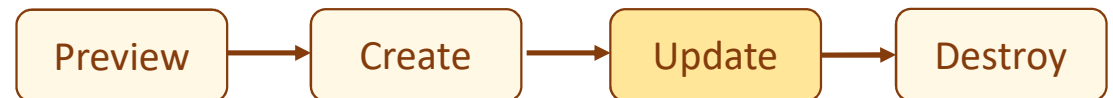
Warning!

It is not guaranteed that if “*terraform plan*” returns no errors that the “*terraform apply*” will necessarily work!

If you attempt to do something that the provider doesn't support or allow, then the apply will fail even though the plan output doesn't report any errors!

Terraform Workflow - apply

The configuration may be updated and “*terraform apply*” run again to **update** the resources to the new *desired* state

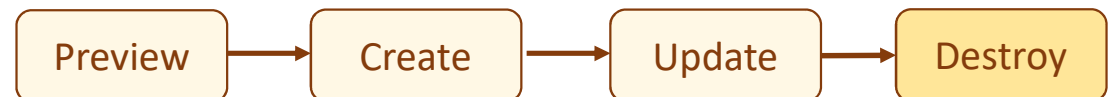


Terraform Workflow - destroy

To destroy resources, use the ***destroy*** command.

```
> terraform destroy
aws_instance.example: Refreshing state... (ID: i-f3d58c70)
aws_elb.example: Refreshing state... (ID: example)
aws_elb.example: Destroying...
aws_elb.example: Destruction complete
aws_instance.example: Destroying...
aws_instance.example: Destruction complete

Apply complete! Resources: 0 added, 0 changed, 2 destroyed.
```



<https://www.terraform.io/docs/commands/destroy.html>

@mjbright CONSULTING

Terraform – AWS Setup

You will need to export your AWS keys as environment variables.

On Mac or Linux, do this with these commands:

```
export AWS_ACCESS_KEY_ID=<access_key>  
export AWS_SECRET_ACCESS_KEY=<secret_key>
```

On Windows, use set instead of export.

You can change the region in the Terraform code if desired.

Lab 1 – Introduction to Terraform

In this lab we will create an initial configuration to launch a single AWS EC2 virtual machine instance.

We will preview, then create this configuration following the basic Terraform workflow

We will then enable ssh access to show that we can have ssh access to the instance

We will use the “terraform graph” command to create a graphical representation of the Terraform resources we created

Quiz

What does “*terraform init*” do ?

When was Terraform created ?

What happens when you type “*terraform apply*” ?

Terraform Templates

Parameterizing Terraform templates

We can parameterize our templates using *variables*.

Description, default and type attributes are optional.

```
variable "name" {  
    Description = "The name of the EC2 instance"  
}
```

Parameterizing Terraform templates

Note: the use of the `${}` syntax to interpolate `var.name`

```
variable "name" {
    description = "The name of the EC2 instance"
}

resource "aws_instance" "example" {
    ami = "ami-408c7f28"
    instance_type = "t2.micro"
    tags {
        name = var.name           # simplified terraform 0.12 syntax
        info = "${var.name} info" # Required syntax for complex interpolation
    }
}
```

Submitting a value

Note: in the previous example we didn't set a value for the variable, only its description.

When run with “*plan*” or “*apply*” will be prompted for the value

```
> terraform plan
var.name
  Enter a value: foo

~ aws_instance.example
  Tags.Name: "terraform-example" => "foo"
```

Submitting a value

We can also pass values to variables by

- using the -var parameter on the command line or
- Setting environment variable TF_var_foo

```
> terraform apply -var name=foo
aws_instance.example: Refreshing state...
aws_instance.example: Modifying...
tags.Name: "terraform-=example" => "foo"
aws_instance.example: Modifications complete.

Apply complete! Resources: 0 added, 1 changed, 0 destroyed.
```

Submitting a value

Variables can be set:

- By their “*default*” value field
- By the `–var` command-line option
- By environment variables
- By prompt (Terraform will prompt for unassigned variables)
- From `terraform.tfvars` or `*.auto.tfvars` files if present

<https://www.terraform.io/docs/configuration/variables.html>

Lab 2 – Terraform Variables

In this lab we will see how we can parametrize the configuration template using variables

Quiz

When would you use `${}` syntax, and when not ?

What are the 5 ways you can define variables for Terraform ?

Terraform Types

Terraform variables

Terraform supports different types of variables

- Strings
- Booleans
- Lists
- Maps
- Sets
- any, null

<https://www.terraform.io/docs/configuration/types.html>

<https://www.terraform.io/docs/configuration/expressions.html>

Terraform variables

The standard way to create a variable is by simply declaring it like so:

```
variable "myvar" {  
    default = "some_value"  
    description = "some_description"  
}
```

Both the default and descriptions are optional.

Terraform strings

Simple strings are enclosed in “ ” quotes:

```
myvar = "a string"
```

Terraform supports multiline strings:

```
template = <<-EOF
    #!/bin/bash
    run-microservice.sh
    EOF
}
```

Terraform arithmetic

Terraform also supports basic arithmetic operations.

- + - * / % for integers
- + - * / for floats.

```
variable "myvar" {  
    default = 1 + 1  
    description = "one plus one"  
}
```

<https://www.terraform.io/docs/configuration/expressions.html#arithmetic-and-logical-operators>

Terraform variables

Terraform supports lists:

```
variable "mylist" {  
    type = "list"  
    default = ["foo", "bar", "baz"]  
}  
  
Variable "mylist2" {  
    default = mylist  
}
```

Note that type is optional.

Terraform can figure out that it's a list from the default syntax.

Terraform lists

We can access individual elements in the list by using the *element()* function like so, to obtain the 0th element of a list:

```
variable "mystring" {  
    default = element(mylist, 0)  
}  
  
variable "mystring2" {  
    default = mylist.0  
}  
  
variable "mylist2" {  
    default = mylist.*  
}
```

Terraform lists

We can also access list elements directly like so:

```
value = "${my_list.0}"
```

Or

```
value = mylist.0
```

This will return the zeroth element of my_list to the value variable.

We can also use wildcards to obtain the whole list:

```
values = "${my_list.*}"
```

or

```
values = my_list.*
```


Terraform Control Structures

Terraform if statements.

As of today (Nov 2019), Terraform doesn't natively support if/elif/else decision trees.

You have to be creative.

For example, using count with a boolean variable.

- Terraform does support the ?: ternary operator

```
resource "aws_route53_health_check" "service_up" {  
    count = var.is_internal_alb ? 0 : 1  
    ...  
}
```

Terraform loops

We can use the **count** function to perform looping.

For example if we have the following:

```
resource "aws_subnet" "vpc_subnets" {  
    <params defined here>  
    count = length(var.vpc_subnet_cidr)  
    cidr_block = element(var.vpc_subnet_cidr, count.index)  
    tags = {  
        Name = "subnet-${count.index+1}"  
    }  
}
```

This will create several instances of this resource.

Note the use of **count.index** to identify the loop number (starts at 0)

Terraform loops

We can also loop over terraform lists:

```
count = length(some_list)
```

The length function returns the number of elements in the list.

Note that elements begin with zero, not one!

We can access the current value in the count with count.index

Resource dependencies.

We can create dependencies between resources.

i.e. we can require that resource B requires the existence of resource A.

Resource dependencies

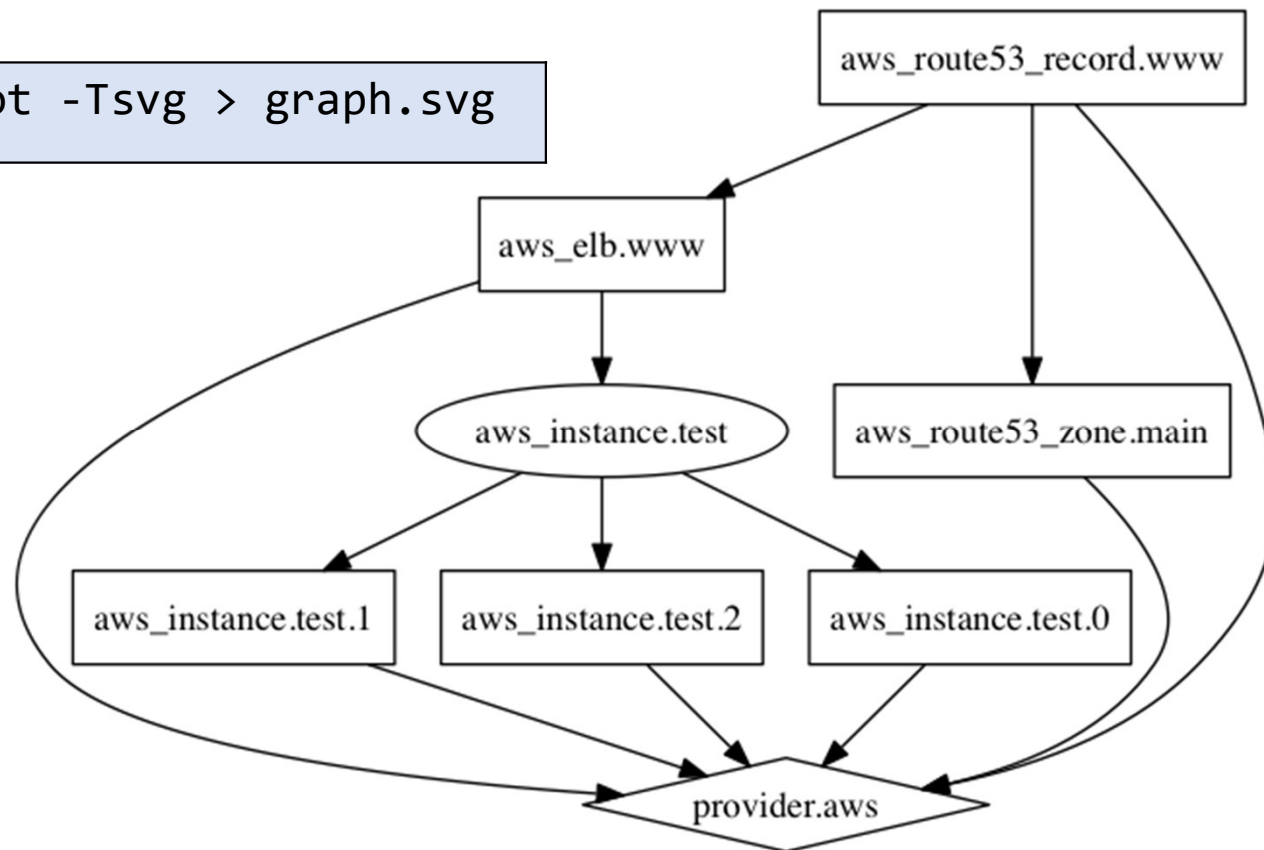
Notice the use of a resource id to specify the dependence

```
resource "aws_eip" "example" {  
    instance aws_instance.example.id  
}  
  
resource "aws_instance" "example" {  
    ami = "ami-408c7f28"  
    instance_type = "t2.micro"  
    tags { Name = var.name }  
}
```

<https://github.com/hashicorp/terraform/blob/master/website/intro/getting-started/dependencies.html.md>

Terraform graph

```
terraform graph | dot -Tsvg > graph.svg
```



<https://www.terraform.io/docs/commands/graph.html>

@mjbright CONSULTING

Lab 3 – Terraform Lists

In this lab we will see how we can use variables of type list

- Setting of count to the length of the vpc_subnet_cidr list
- Setting individual cidr_block attribute of aws_instances to the appropriate cidr

Quiz

What is special about the count variable ?

Can you loop 10 times with Terraform ?

What would happen if we change the value of count, then re-applied ?

Terraform maps

Maps are key/value pairs.

Terraform supports declarations:

```
variable "mylist" {  
  type = "map"  
  default = {  
    "foo" = "bar",  
    "baz" = "blech"  
  }  
}
```

Terraform functions

- Numeric: min, max
- String: chomp, join, split, replace, substr, upper, lower, ...
- List: length, sort, element, count (allows looping over list)
- Maps: lookup (looks up a map value based on a key)
- Filesystem: file, fileexists, dirname, ...
- Date/time: formatdate, timeadd, timestamp
- Hash/crypto: filemd5, filesha256, filesha512, md5, ...
- IP/network: cidrhost, cidrnetmask, cidrsubnet
- Type conversion: tobool, tolist, tomap, toset, tonumber, tostring

<https://www.terraform.io/docs/configuration/functions.html>

Lab 4 – Terraform Maps

In this lab we will see how we can use variables of type map

- Using a map of regions to specify the ami image to use
- Using a map of regions to availability zones

Quiz

What is a possible use case of Maps ?

Terraform Data Sources

Terraform data sources

To get data directly from the cloud provider data sources.

```
data "aws_ami" "example" {  
  most_recent = true  
  
  owners = ["self"]  
  tags = {  
    Name     = "app-server"  
    Tested  = "true"  
  }  
}
```

<https://www.terraform.io/docs/providers/aws/>

Follow "Provider Data Sources" or "EC2"->"Data Sources"

Terraform data sources

Note that we can declare “filters” to only get relevant data from the data source.

Note that “most_recent” is a boolean.

Example:

“Find the latest available AMI that is tagged with Component = web”

```
data "aws_ami" "web" {  
  filter {  
    name     = "state"  
    values   = ["available"]  
  }  
  
  filter {  
    name     = "tag:Component"  
    values   = ["web"]  
  }  
  
  most_recent = true  
}
```


Lab 5 – Terraform Data Sources

In this lab we will see how we can access Provider data sources

Quiz

What is a possible use case of Data Sources ?

Terraform State

Terraform states

Terraform records the state locally in .tfstate json files by default:

```
> cat terraform.tfstate      # some lines removed
{ "version": 4,
  "terraform_version": "0.12.12", {
    "mode": "managed",
    "type": "aws_instance",
    "name": "example",
    "provider": "provider.aws",
    "instances": [ {
      "schema_version": 1,
      "attributes": {
        "ami": "ami-0e81aa4c57820bb57",
        "arn": "arn:aws:ec2:us-west-1:568285458700:instance/i-0f70aa9654b216df5",
        "associate_public_ip_address": true,
        "availability_zone": "us-west-1a",
        "cpu_core_count": 1,
```

Terraform states

The previous state is saved as `terraform.tfstate.backup`

So after a “`terraform destroy`” you can still investigate the state of the previous configuration

The “`terraform show`” command allows to see a human (.tf) readable version of the state

Coordinating Terraform states

When working in a team shared access to the Terraform state is necessary so you need

- Shared file system
- Locking
- Secrets

Enabling remote state storage

The simplest “*remote backend*” solution when running on AWS is S3

It supports encryption, locking via DynamoDB, and versioning

It is possible to enable S3 storage to store terraform.tfstate files:

```
> terraform remote-config \  
  -backend=s3 \  
  -backend-config=bucket=my-s3-bucket  
  -backend-config=key=terraform.tfstate  
  -backend-config=encrypt=true  
  -backend-config=region=us-east-1
```

Coordinating Terraform states

Hashicorp provides the *Atlas* / Terraform Cloud service which can store terraform.tfstate files.

It provides file locking, but it is expensive.

You can also create a Continuous Integration job manually with a tool like Jenkins.

Another good alternative is to use Terragrunt.

Terraform remote state data source

```
data "terraform_remote_state" "db" {  
  backend = "s3" config = {  
    # Replace this with your bucket name!  
    bucket = "terraform-up-and-running-state"  
    key = "stage/data-stores/mysql/terraform.tfstate"  
    region = "us-east-2"  
  }  
}
```

<https://blog.gruntwork.io/how-to-manage-terraform-state-28f5697e68fa> "How to manage Terraform State"

Terraform State - Workspaces

Workspaces: allow to change context and create new resources isolated from resources in another context

“terraform workspace” allows to manage workspaces

- List
- Create
- Destroy

Warning: The workspace does not sufficiently protect the state when working in a multiple team member environment

<https://www.terraform.io/docs/state/workspaces.html>

Lab 6 – Storing Persistent States

In this lab we will see how to store Terraform state information in a remote location such as an S3 bucket or a database

Quiz

Why is it a good idea to store state in a remote backend ?

What are examples of state backends ?

Terragrunt

Terragrunt is an open source wrapper for Terraform.

- Provides locking vs. DynamoDB.
- Looks for its configuration with a .terragrunt file.

The next example shows a sample .terragrunt file.

<https://blog.gruntwork.io/how-to-manage-terraform-state-28f5697e68fa> “How to manage Terraform State”

Sample terragrunt file.

Here's an example of enabling remote storage of terraform state files.

```
dynamoDbLock = {  
  StateFileID = "mgmt/bastion-host"  
}  
remoteState = {  
  backend = "s3"  
  backendConfigs = {  
    bucket = "example-co-terraform-state"  
    key = "mgmt/basion-host/terraform.tfstate"  
  }  
}
```

Terragrunt

Simply replace the terraform command with terragrunt to run commands like plan, apply and destroy.

Terragrunt automatically obtains and releases locks on apply and destroy commands.

Example of terragrunt apply

```
> terragrunt apply
[terragrunt] Acquiring lock for bastion-host in DynamoDB
[terragrunt] Running command: terraform apply

aws_instance.example: Creating...
  ami:                                "" => "ami-0d729a60"
  instance_type:                      "" => t2.micro
[...]
Apply complete! Resources: 1 added, 0 changed, 0 destroyed.

[terragrunt] Releasing lock for basion-host in DynamoDB.
```


Terraform Modules

Terraform Modules

Terraform modules are directories that contain one or more terraform templates.

We can re-use these modules and templates, and subject them to version control.

Terraform Modules

By convention we define three specific terraform templates in a module.

- vars.tf
- main.tf
- outputs.tf

Vars.tf example

Specify module inputs in vars.tf:

```
variable "name" {  
    description = "The name of the EC2 instance"  
}  
  
variable "ami" {  
    description = "The AMI to run on the EC2 instance"  
}  
  
variable "port" {  
    description = "The port to listen on for HTTP requests"  
}
```

main.tf example

Specify resources in main.tf:

```
resource = "aws_instance" "example" {  
    ami          = var.ami  
    instance_type = "t2.micro"  
    user_data     = template_file.user_data.rendered  
    tags { Name = var.name }  
}
```

Outputs.tf

Specify outputs in outputs.tf:

```
output "url" {  
    value = "http://${aws_instance.example.ip}:${var.port}"  
}
```

Outputs are reported in the terraform.tfstate

Terraform modules

Note that terraform modules don't have "scope".
Terraform modules don't share variables implicitly.

You must define inputs

- in the module vars.tf, and reference these from the root module

You must define outputs

- in the module resources.tf, so that other modules can use them.

Modules are like "functions" in other programming languages.

Terraform modules

Terraform configs always have a 'root' module – the working directory where *“terraform apply”* is run

You 'get' modules with *“terraform get”* or *“terraform init”*.

Module locations can be specified with the source keyword.

Note that region is defined here as an input variable to module “vpc_module”

```
module "vpc_module" {  
  source = "./modules/vpc"  
  region = var.region  
}
```


Terraform modules

- Module sources can be:
 - Local files
 - Git repositories
 - URL's
 - Terraform registry locations.
 - Bitbucket
 - Mercurial repositories
 - Other ...

Terraform modules

For variables to be accessible in the calling module, they must be defined as outputs in the called module

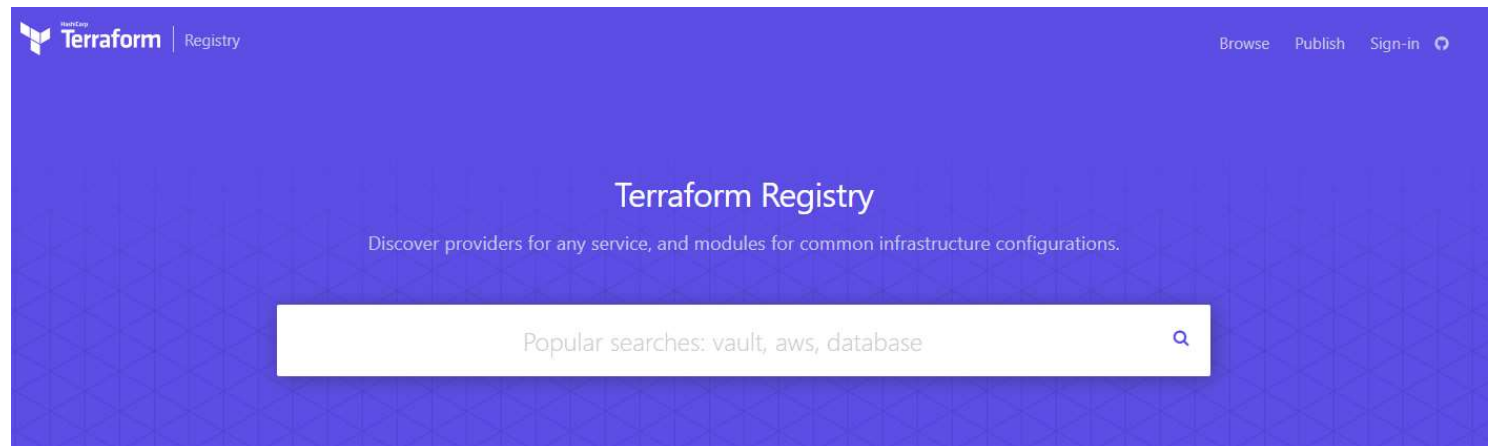
In the module: ./modules/vpc/output.tf

```
output "subnet_ids" {  
    value = aws_subnet.vpc_subnets.*.id  
}
```

In the “calling” module: modules/instances/main.tf

```
"aws_instance" "webserver" {  
    count      = length(module.vpc_module.aaz[var.region])  
    ami        = lookup(var.ami_instance, var.region)  
    subnet_id  = element(module.vpc_module.subnet_ids, count.index)
```


Terraform Registry - Modules



Find Terraform Modules


Use and learn from verified and community modules

POPULAR MODULES



security-group
aws

Terraform module which creates EC2-VPC security groups on AWS



vpc
aws

Terraform module which creates VPC resources on AWS

[@mjbright CONSULTING](#)

<https://registry.terraform.io/>

Terraform Registry - Modules

Modules can be sourced from the Terraform public registry

Or from the Terraform Cloud private registry (paid service)

Or from any service that implements the registry API

<https://www.terraform.io/docs/registry/api.html>

<https://registry.terraform.io/>

Lab 7 – Terraform Modules

In this lab we will see how we can split Terraform code into modules.
More importantly we will see the advantages of modules for code management,
testing, sharing

Quiz

By convention what are the three files which make up a module ?

Modules are like functions, so how are their input/output specified ?

Terraform Provisioners

Terraform Provisioners

Provisioners are not the same thing as providers !!

Provisioners are typically used with Terraform to run remote commands on the created resource - provided that the resource supports it.

- Mainly used to run resource configuration management.
- Provisioners are added to resource definitions.

Terraform Provisioners

They are provided for pragmatism and should generally be avoided as Terraform cannot predict their effects !

Can be used to model specific actions on the local machine or remote machine in order to prepare servers or other infrastructure objects for service.

A good use case would be to prepare (local hosts file) and execute **configuration management tools** which affect the internal state of resources

<https://www.terraform.io/docs/provisioners/>

@mjbright CONSULTING

Terraform Provisioners

Several in-built Provisioners exist:

- file
- habitat
- local-exec
- remote-exec
- chef
- puppet
- salt-masterless

<https://www.terraform.io/docs/provisioners/>

Terraform Provisioners

Local-exec performs an action on the local machine:

```
resource "aws_instance" "web" {  
  # ...  
  
  provisioner "local-exec" {  
    command = "echo ${self.private_ip} > file.txt"  
  }  
}
```

One use case might be to build up an `ansible_hosts` file

<https://www.terraform.io/docs/provisioners/local-exec.html>

<https://ilhicas.com/2019/08/17/Terraform-local-exec-run-always.html>

@mjbright CONSULTING

Terraform Provisioners

Many provisioners are remote, and will require specific information to be supplied in order to connect.

<https://sdorsett.github.io/post/2018-12-26-using-local-exec-and-remote-exec-provisioners-with-terraform/>

Terraform Provisioners

Example provisioner, “file”, used to copy files between local and remote systems:

```
provisioner "file" {  
  source      = "conf/myapp.conf"  
  destination = "/etc/myapp.conf"  
  
  connection {  
    type      = "ssh"  
    user      = "root"  
    password  = "${var.root_password}"  
  }  
}
```

<https://www.terraform.io/docs/provisioners/file.html>

Terraform Provisioners

Example provisioners
to copy then
execute scripts
on remote system:

```
provisioner "file" {  
    source      = "setup.sh"  
    destination = "setup.sh"  
}  
  
provisioner "remote-exec" {  
    inline = [ "chmod +x ./setup.sh", "./setup.sh" ]  
}  
  
connection {  
    type = "ssh"  
    host = self.public_ip  
    user = "ubuntu"  
    private_key = file(pathexpand("~/ssh/id_rsa"))  
}
```

Terraform Provisioners

Example provisioner to execute script on local system:

```
provisioner "local-exec" {  
  command = <<-EOF  
    echo "PRIVATE_IP=${aws_instance.example.private_ip}" >> ips.txt  
    echo "PUBLIC_IP=${aws_instance.example.public_ip}" >> ips.txt  
    echo "PUBLIC_DNS=${aws_instance.example.public_dns}" >> ips.txt  
  EOF  
}
```

This could be the basis for building up an `ansible_hosts` file

Lab 8 – Setting up Web Servers

In this lab we will see how we can launch several Web Servers using Terraform

Quiz

What are provisioners ?

Why should you avoid using them ?

When might you use them ?