Introduction to Terraform (2d)

Michael Bright / @mjbright Consulting

https://linkedin.com/in/mjbright @mjbright @mjbright

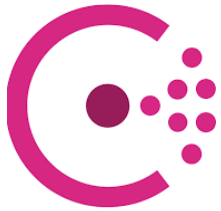
Introduction

Introduction

HashiCorp
Packer

HashiCorp
Vagrant

HashiCorp
Nomad



HashiCorp
Consul

HashiCorp
Terraform

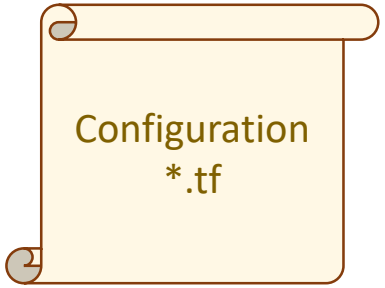
HashiCorp
Vault

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.



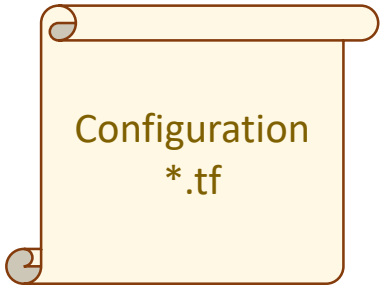
Introduction

"Infrastructure as Code" (IaC) can be achieved through

- Adhoc scripting — Shell, Python, Perl, …
- Config managemt 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, Audit trail
- 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

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 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*" ?

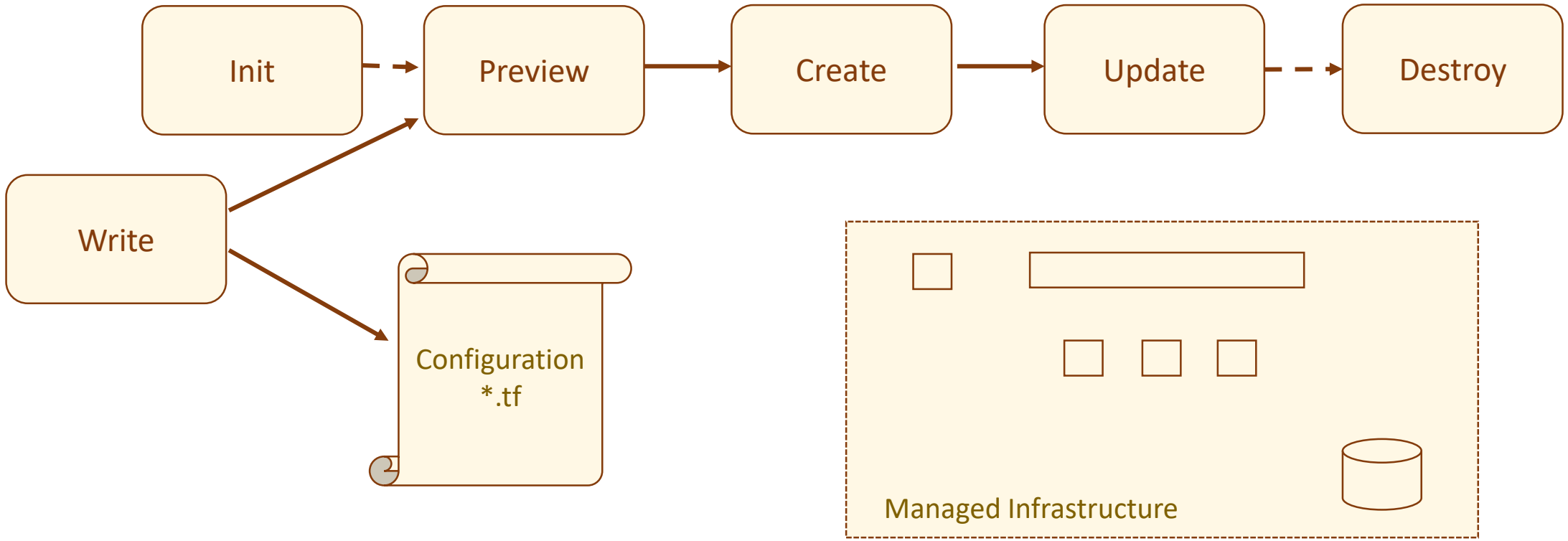
Is Terraform an Open Source project ?

What company is behind Terraform ?

Terraform Workflow

Terraform Workflow

Use of Terraform is according to the workflow

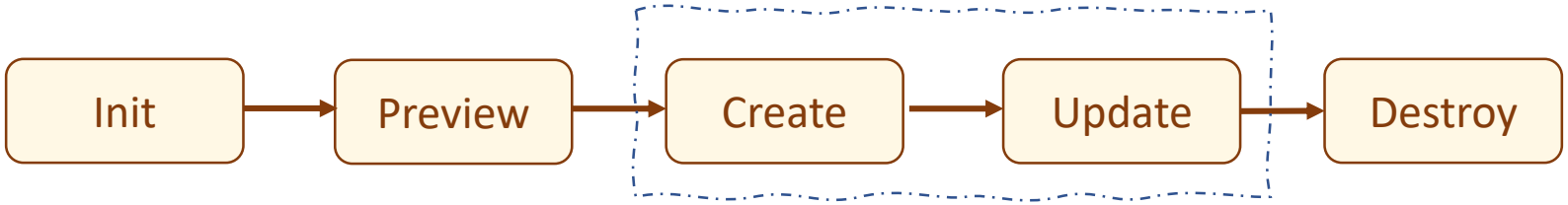


https://www.terraform.io/docs/commands/index.html

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

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=~/dot.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

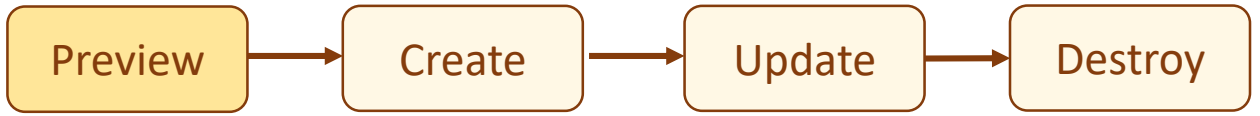
The *plan* command creates an execution plan.

```
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.
```

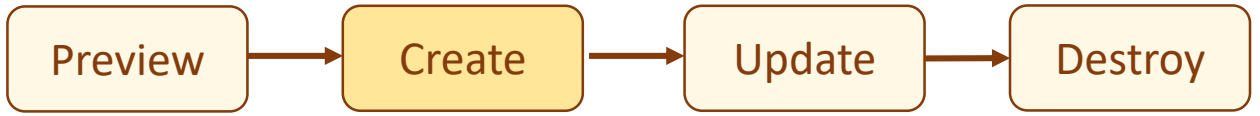
https://www.terraform.io/docs/commands/plan.html

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

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

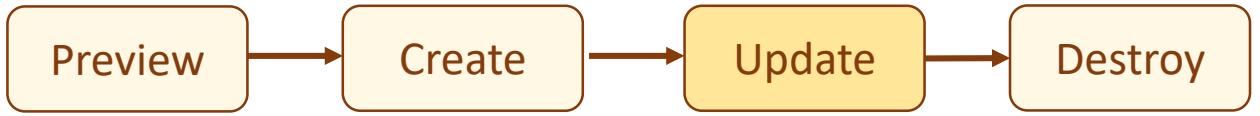
Warning!

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

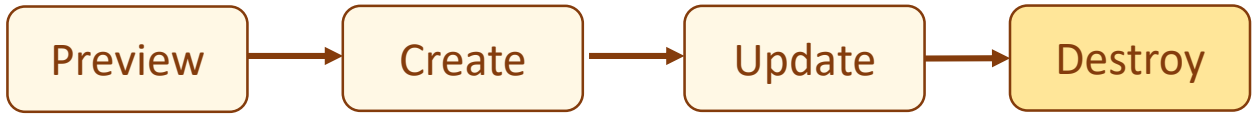


Terraform Workflow - destroy

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

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.

AWS_DEFAULT_REGION may be exported to provide a default region value

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*" we will be prompted for the value

```
> terraform plan
The name of the EC2 instance
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 the –var-file <file> command-line option
- By environment variables of the form TF_VAR_variable for "variable"
- From terraform.tfvars or *.auto.tfvars files if present
- If a variable is unassigned at plan/apply time - Terraform will prompt for a value

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

Submitting a value

Precedence for variables

- If the default value for the variable is defined it is used
- If TF_VAR_<var> is set e.g. TF_VAR_varname this overrides
- If --var var=val is passed as an argument this takes precedence

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

Submitting a value

Precedence for the region attribute/variables

- If region variable is unset & AWS_DEFAULT_REGION is set, it is used
- If the default value for region is defined it is used
- If TF_VAR_region is set, this overrides
- If --var var=val is passed as an argument this takes precedence

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 “${variable}” syntax, and when not ?

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

Terraform Types

Terraform Types

Terraform supports different types of variables

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

https://www.terraform.io/docs/configuration/types.html
https://www.terraform.io/docs/configuration/expressions.html

Terraform Types: Simple Types

Terraform supports different types of variables

Simple Types:

- string: a sequence of Unicode characters e.g. "hello“
- number: a numeric value. Integer or floating point
- boolean: true or false. bool values can be used in conditional logic

https://www.terraform.io/docs/configuration/types.html
https://www.terraform.io/docs/configuration/expressions.html

Terraform Types: Simple Types

The standard way to create a variable is by simple declaration:

```
variable “mystr” {
         default = “some_value”
         description = “just a string”
}
```

Both the default and descriptions are optional

Terraform recognizes the type as “string” in this case

```
variable “mynum” {
         default = 23.7
         description = “just a number”
}

variable “mybool” { default = true }
```

Terraform Types: Strings

Simple strings are enclosed in “ “ quotes:

```
myvar = “a string”
```

Terraform supports multiline strings:

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

Terraform Types: Number arithmetic

Terraform also supports basic arithmetic operations.

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

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

output "even_1_1_3_mod2" { value = (1 + 1 + 3) % 2 == 0 } # false
output "even_1_1_4_mod2" { value = (1 + 1 + 4) % 2 == 0 } # true
```

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

Terraform Types: Boolean

Terraform also supports booleans

```
variable mytruth {
         default = true
         description = "is this really true?"
}

output mytruth          { value = var.mytruth }
output is_mytruth_true { value = (var.mytruth == true) }
output bool_true        { value = true }
output not_bool_true    { value = !true }
```

Outputs:

```
_true          = true
is_mytruth_true = true
mytruth        = true
not_bool_true  = false
```

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

Terraform Types : Simple Types

Conversion of Primitive Types

Terraform automatically converts number & bool values to string values as needed, & vice-versa as long as the string contains a valid representation of a number or boolean value

- true converts to "true", and vice-versa
- false converts to "false", and vice-versa
- 15 converts to "15", and vice-versa

https://www.terraform.io/docs/configuration/types.html
https://www.terraform.io/docs/configuration/expressions.html

Terraform Types : Collections

Complex Types: Collections

- list: sequence of values – all the same type - indexed from zero.
- map: collection key/value pairs, where keys are strings
- set: collection of unique values

https://www.terraform.io/docs/configuration/types.html
https://www.terraform.io/docs/configuration/expressions.html

Terraform Types: Collections - Lists

Terraform supports lists:

```
variable “mylist” {
        type = “list”  # not needed
        default = [“foo”, “bar”, “baz”]
}
```

Note that type is optional.

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

Terraform Types: Collections - Lists

We can access individual elements in the list by using the *element()* function, or with other syntaxes:

```
variable mylist {
        type = list(number) # unneeded
        default = [1,2,3]   # All elements must be of same type
}

output elem0  { value = element(var.mylist, 0) }
output elem1  { value = element(var.mylist, 1) }
output elem1a { value = var.mylist.1 }
output elem1b { value = var.mylist[1] }

output list   { value = var.mylist }
output lista  { value = var.mylist.* }
```

Outputs:

elem0 = 1
elem1 = 2
elem1a = 2
elem1b = 2

list = [1, 2, 3,]
lista = [1, 2, 3,]

Terraform Types: Complex - Object

Complex Types: Structural Types

- object: collection of named attributes that each have their own type.

  { <KEY> = <TYPE>, <KEY> = <TYPE>, ... }

For example: an object type of object({ name=string, age=number }) would match a value like the following:

```
{
   name = “John”
   age    = 52
}
```

https://www.terraform.io/docs/configuration/types.html
https://www.terraform.io/docs/configuration/expressions.html

Terraform Types: Complex - Object

```
variable address_johndoe {
   type=object({name=string,age=number})
   default={name="john doe", age=33}
}

variable addressbook {
   type=list(object({name=string,age=number}))
   default=[ {name="wilson smith", age=34} ]
}

output addressbook { value = var.addressbook }
output addressbook2 { value = concat(var.addressbook, [var.address_johndoe])}
```

Outputs:

```
addressbook = [   {
   "age" = 34
   "name" = "wilson smith"
 },  ]
addressbook2 = [   {
   "age" = 34
   "name" = "wilson smith"
 },  {
   "age" = 33
   "name" = "john doe"
 },   ]
```

https://www.terraform.io/docs/configuration/types.html
https://www.terraform.io/docs/configuration/expressions.html

Terraform Types: Complex - Tuple

Complex Types: Structural Types

- tuple: sequence of elements identified by consecutive whole numbers starting with zero, where each element has its own type.

a tuple type of tuple ([string, number, bool]) would match a value like the following:

["a", 15, true]

https://www.terraform.io/docs/configuration/types.html
https://www.terraform.io/docs/configuration/expressions.html

Terraform Types: Complex - Tuple

```
variable atuple {
   type=tuple([string,number,bool])
   default=["john doe", 33, true]
}

output atuple { value = var.atuple }
output atuple0 { value = var.atuple[0] }
output atuple1 { value = var.atuple.1 }
output atuple2 { value = var.atuple.2 }
```

Outputs:

```
atuple = [
  "john doe",
  33,
  true,
]
atuple0 = john doe
atuple1 = 33
atuple2= true
```

https://www.terraform.io/docs/configuration/types.html
https://www.terraform.io/docs/configuration/expressions.html

Terraform Control Structures

Terraform Control Structures: loops

We can also “loop” over terraform lists:

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

When the special attribute “count” is specified in a resource multiple resources are created.

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

Note that elements begin with zero, not one!

Terraform Control Structures: loops

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

For example if we have the following:

```
resource "aws_subnet" "vpc_subnets" {
        # lines removed …
        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 if statements.

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

You have to be creative.

We can use count with a boolean variable and the "?" ternary operator

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

Resource dependencies.

By studying the resource attributes used in interpolation expressions, Terraform automatically infers when one resource depends on another.

But there are cases where the dependency has to be specified – see example …

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 }
}
```

Terraform uses the dependency information to determine the correct order in which to create resources.

In this example, we create an *implicit dependency* so that Terraform knows to create the *aws_instance* before the *aws_eip*.

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

Resource dependencies

Notice the use of a resource id to specify the dependence

```
resource "aws_s3_bucket" "example" { # S3 bucket for our application
  bucket = <unique bucket name>
  acl    = "private"
}

resource "aws_instance" "example" {
  ami           = "ami-2757f631"
  instance_type = "t2.micro"

  # This EC2 instance to be created only after S3
  depends_on = [aws_s3_bucket.example] # explicit dependency
}
```

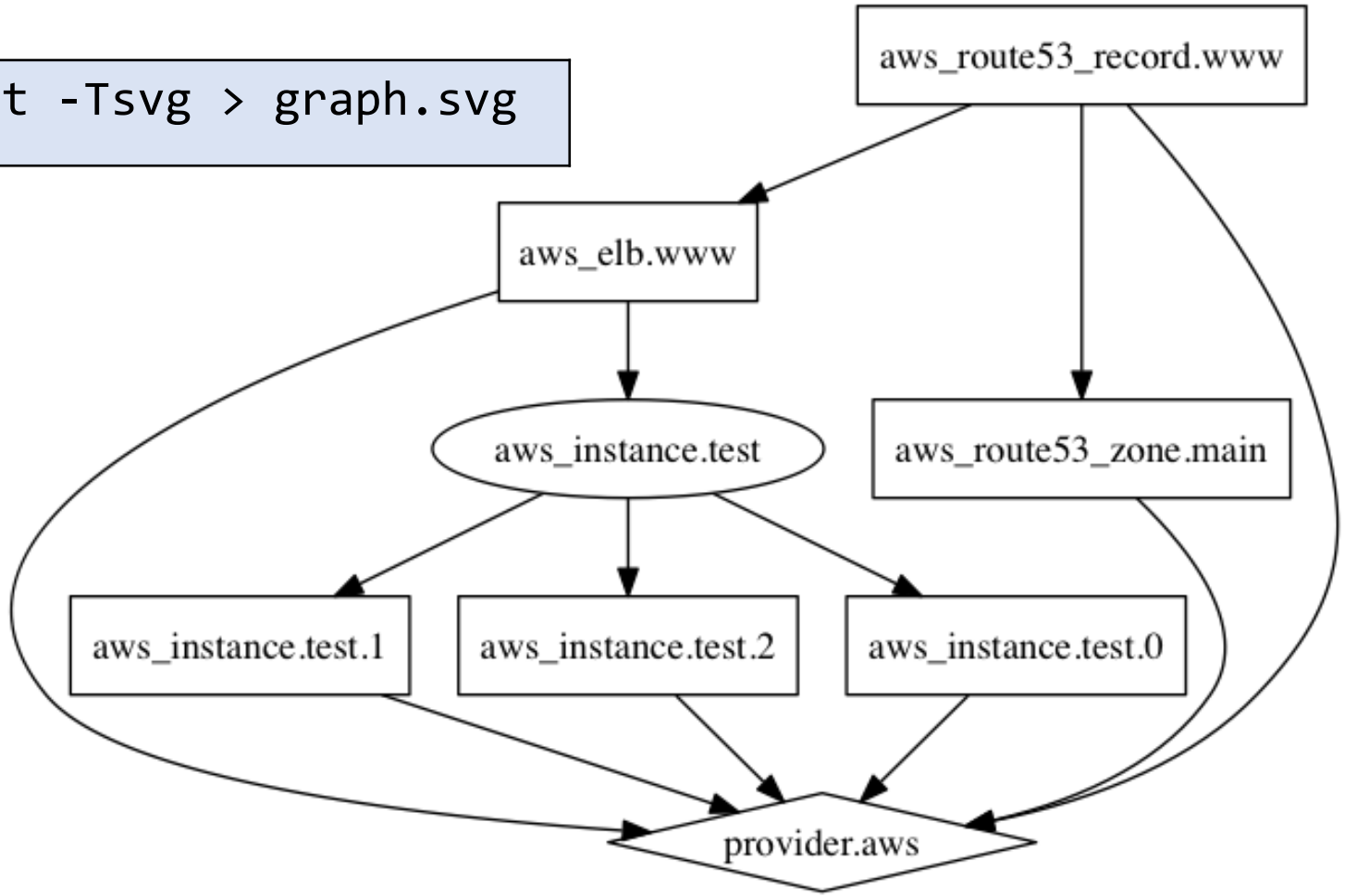
Sometimes there are dependencies between resources that are not visible to Terraform.

The *depends_on* argument is accepted by any resource and accepts a list of resources to create explicit dependencies for

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

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, filexists, 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

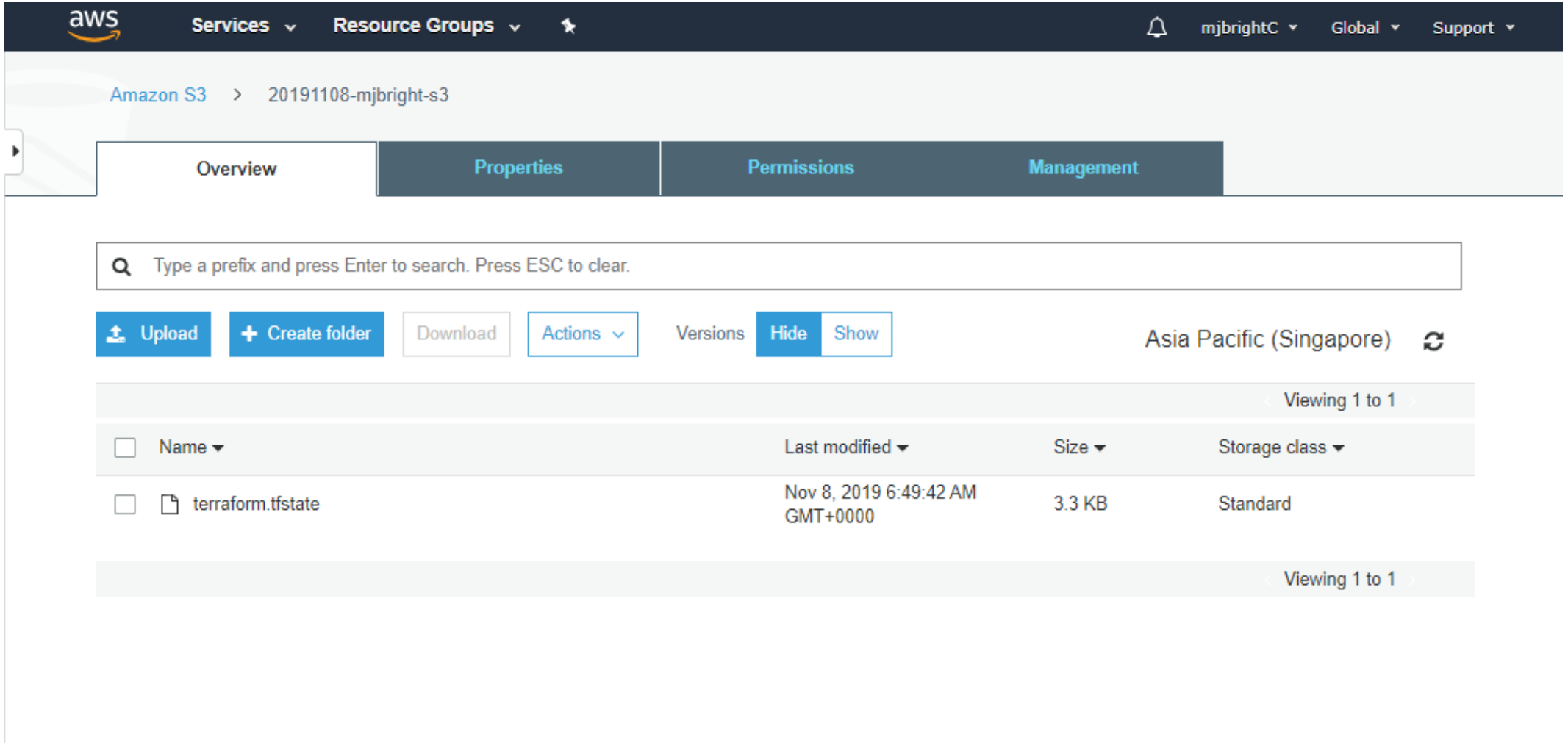
Create a backend.tf containing:

```
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"
}
```

Perform a “terraform init” to enable the remote storage backend

https://blog.gruntwork.io/how-to-manage-terraform-state-28f5697e68fa “How to manage Terraform State”
http://aws-cloud.guru/terraform-aws-backend-for-remote-state-files-with-s3-and-dynamodb/

Terraform remote state data source



Terraform remote state data source


```
{
    "version": 4,
    "terraform_version": "0.12.13",
    "serial": 0,
    "lineage": "052409a2-dfb2-0359-852f-971991c6c4c8",
    "outputs": {
        "dynamodb_table_name": {
            "value": "20191108-mjbright-dynambodb-lock",
            "type": "string"
        },
        "s3_bucket_arn": {
            "value": "arn:aws:s3:::20191108-mjbright-s3",
            "type": "string"
        }
    },
    "resources": [
        {
            "mode": "managed",
            "type": "aws_dynamodb_table",
            "name": "terraform_locks",
            "provider": "provider.aws",
            "instances": [
                {
                    "schema_version": 1,
                    "attributes": {
                        "arn": "arn:aws:dynamodb:ap-southeast-1:568285458700:table/20191108-mjbright-dynambodb-lock",
                        "attribute": [
                            {
                                "name": "LockID",
                                "type": "S"
                            }
```

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 some examples of state backends ?

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}”
}
```

https://www.terraform.io/docs/configuration/outputs.html

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

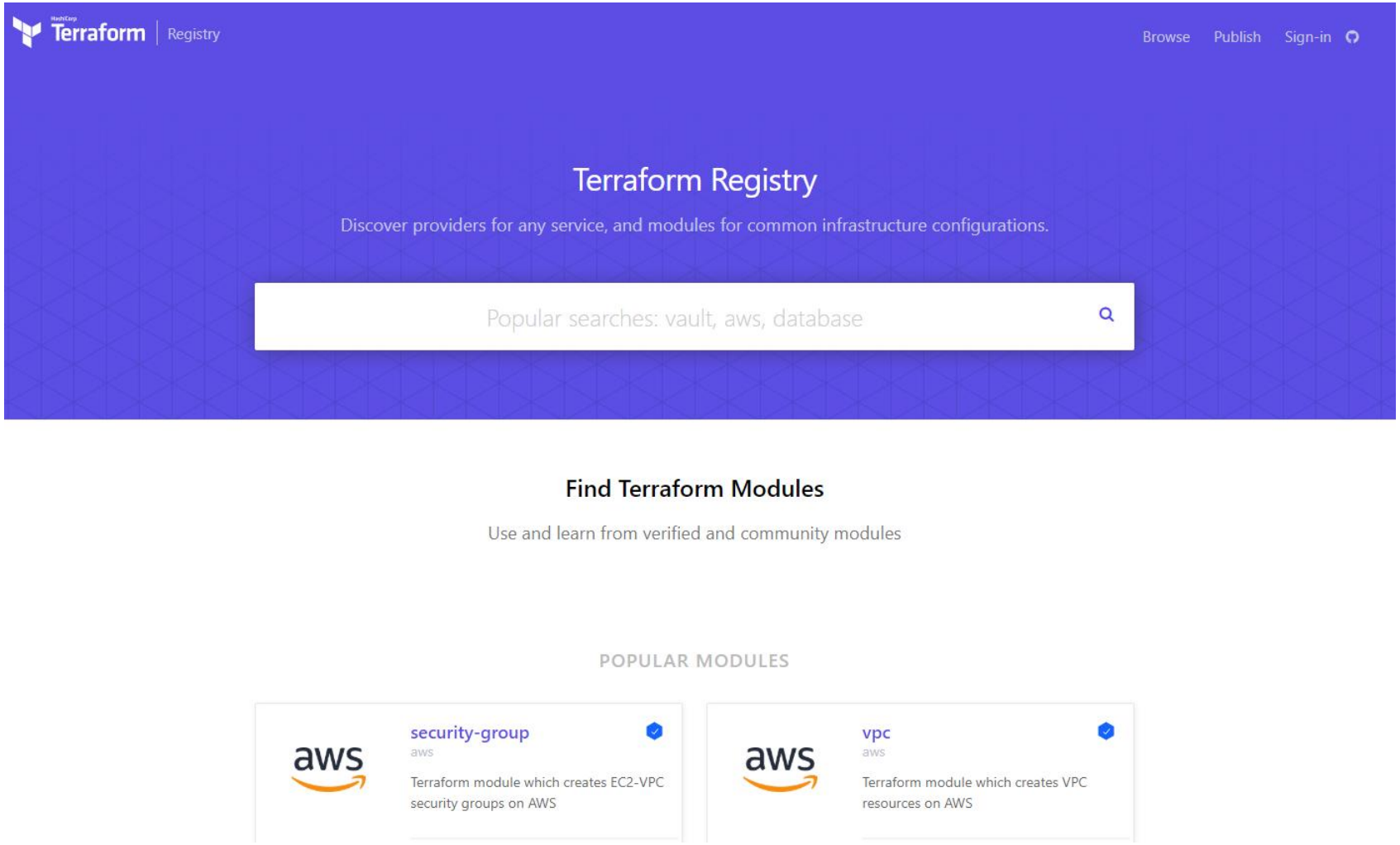
In the module: ./modules/vpc/output.tf

```
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



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/

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

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

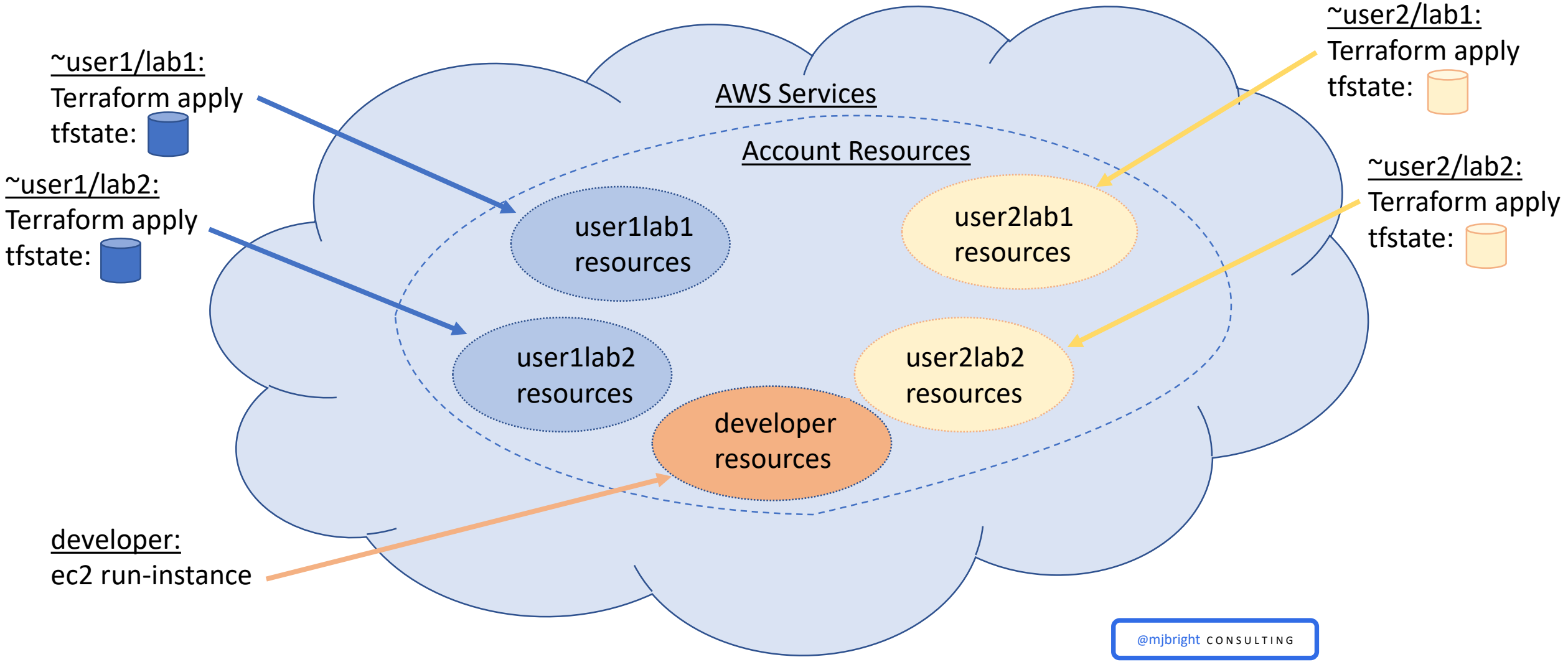
Quiz

What are provisioners ?

Why should you avoid using them ?

When might you use them ?

Terraform – State



Terraform – Importing foreign resources

Terraform import

Terraforming
https://github.com/dtan4/terraforming

Terraform – Import

Terraform import: Allows to import existing resources, from a provider

```
> terraform import aws_instance.example i-abcd1234
```

However, you first need to create a configuration file, e.g. unmanaged.tf with at least ami and instance_type specified:

```
resource "aws_instance" "example" {
    ami = "ami-0e81aa4c57820bb57"
    instance_type = "t2.micro"
}
```

Note: It ignores any resources already created by Terraform

Terraform – Import

Demo of Terraform import

Terraforming – Import

Terraforming: Export AWS resources to Terraform style (tf, tfstate)

http://terraforming.dtan4.net/ https://github.com/dtan4/terraforming

e.g. export tf

```
> terraforming s3
resource "aws_s3_bucket" "hoge" {
    bucket = "hoge" acl = "private“
}
resource "aws_s3_bucket" "fuga" {
    bucket = "fuga" acl = "private“
}
```

e..g export tfstate

```
> terraforming s3 --tfstate
…
```

Terraforming – Import

Terraforming:

is written in ruby, so is a little more complicated to install

But it seems worth the effort if you need to dynamically manage “unmanaged” resources with Terraform

Terraform – AWS Autoscaling & Loadbalancing

Terraform – AWS ELB & ALB

AWS provide 3 types of Load balancer service

Elastic Load Balancer

- ELB: “Classic” Load-balancer, Layer4 routing, TCP, SSL/TLS, HTTP(S)

Application Load Balancer

- ALB: 2nd-gen Load-balancer, Layer7 routing, HTTP, HTTPS

Network Load Balancer

- NLB: 2nd-gen Load-balancer, Layer4 routing, TCP, UDP, TLS

https://medium.com/@sahityamaruvada/setting-up-aws-network-load-balancer-with-terraform-0-12-b87e75992949

For a comparison of AWS Load-balancers:

https://aws.amazon.com/elasticloadbalancing/features/

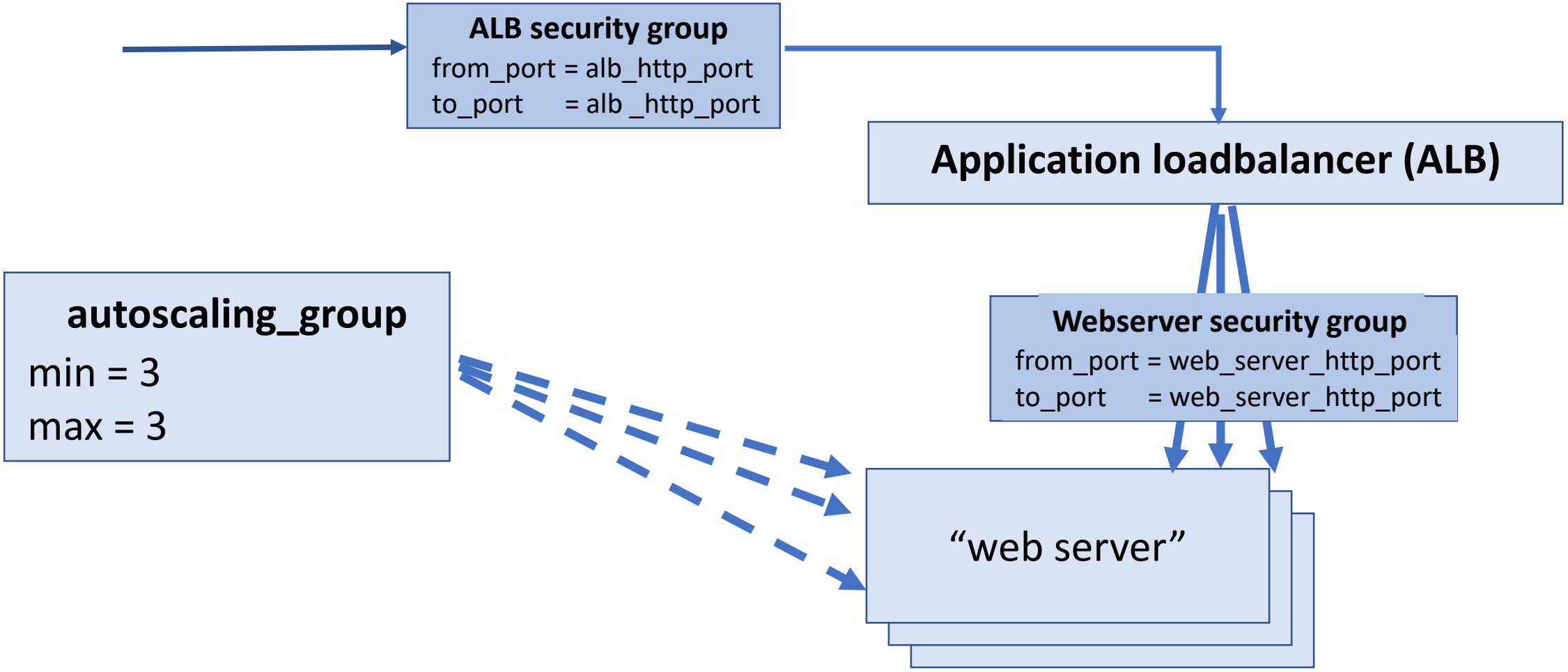
Terraform – AWS Autoscaling & Loadbalancing

To demonstrate the use of AWS Autoscaling

- We will create an AutoScaling group (ASG) to scale the number of instances.

- We will associate this with an Application Load Balancer (ALB), which forwards traffic to the instances

Terraform – AWS Autoscaling & Loadbalancing



Terraform – AWS Autoscaling & Loadbalancing

autoscaling_group

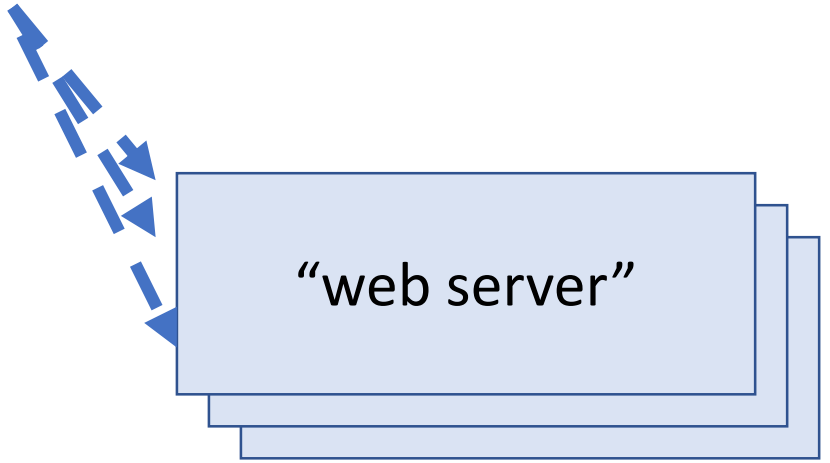
min = 3
max = 3

The AutoScaling group (ASG) scales the number of instances.

The ASG uses a LaunchConfiguration to create the instances.
In this simple example we fix at a constant 3 replicas

launch_configuration

image_id = data.aws_ami.ubuntu.id
instance_type = "t2.micro"




```
lifecycle {
   create_before_destroy = true
}
```

Terraform – AWS Autoscaling & Loadbalancing

```
resource "aws_autoscaling_group" "web_servers" {
   name = aws_launch_configuration.web_servers.name
   launch_configuration = aws_launch_configuration.web_servers.name

  min_size         = 3
  max_size         = 3
  desired_capacity = 3
  min_elb_capacity = 3

  vpc_zone_identifier = data.aws_subnet_ids.default.ids # Deploy all the subnets (& AZs)

  health_check_type = "ELB“  # Register ASG instances in the ALB & use it’s health check

  target_group_arns = [aws_alb_target_group.web_servers.arn]

  depends_on = ["aws_alb_listener.http"]
}
```

```
# To support rolling deployments, must set in all dependencies also
lifecycle {
   create_before_destroy = true
}
```

Terraform – AWS Autoscaling & Loadbalancing

```
resource "aws_launch_configuration" "web_servers" {
  image_id        = data.aws_ami.ubuntu.id
  instance_type = "t2.micro"
  key_name       = var.key_name

  security_groups = [aws_security_group.web_server.id]

  # To keep this example simple, we run a web server as a User Data script.
  # In real-world usage, you would install in the AMI.
  user_data = <<-EOF
              #!/bin/bash
              echo "${var.server_text} from $(hostname)" > index.html
              nohup busybox httpd -f -p "${var.web_server_http_port}" &
              EOF
}
```

```
lifecycle {
    create_before_destroy = true
 }
```

Terraform – AWS Autoscaling & Loadbalancing

ALB

ALB Listener

port = var.alb_http_port
protocol = "HTTP"

ALB Listener Rule

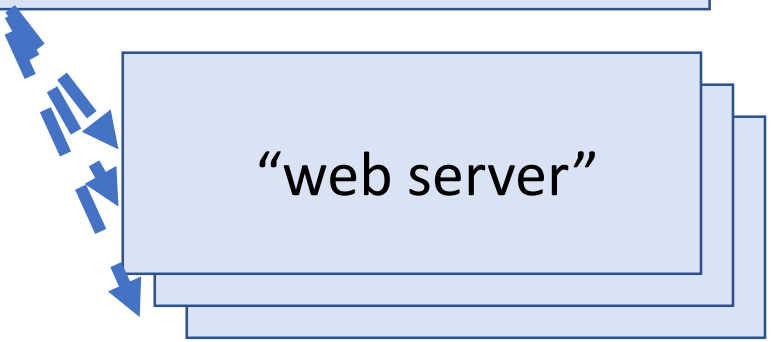
type = “forward”
target_group_arn

ALB Target Group

port = var.web_http_port
Health_check

The ALB has a listener
with a rule to listen on the web_server_http_port.

The ALB forwards traffic to the ALB Target Group




```
lifecycle {
    create_before_destroy = true
}
```

Terraform – AWS Autoscaling & Loadbalancing

```
resource "aws_alb" "web_servers" {
  name            = var.name
  security_groups = [aws_security_group.alb.id]
  subnets         = data.aws_subnet_ids.default.ids
}
```

```
lifecycle {
  create_before_destroy = true
}
```

```
resource "aws_alb_listener" "http" {
  load_balancer_arn = aws_alb.web_servers.arn
  port              = var.alb_http_port
  protocol          = "HTTP"

  default_action {
    type             = "forward"
    target_group_arn = aws_alb_target_group.web_servers.arn
  }
}
```

Terraform – AWS Autoscaling & Loadbalancing

```
resource "aws_alb_target_group" "web_servers" {
  name     = var.name
  port     = var.web_server_http_port
  protocol = "HTTP"
  vpc_id   = data.aws_vpc.default.id

  # Give existing connections 10 secs to complete before deregistering an instance ( default is 300 seconds )
  # In theory, ALB should deregister the instance when no open connections; in practice, it waits full delay.
  # If your requests are processed quickly, set to something lower (e.g. 10 seconds) to keep redeploys fast.
  deregistration_delay = 10

  health_check {
    path                = "/"
    interval            = 15
    healthy_threshold   = 2
    unhealthy_threshold = 2
    timeout             = 5
```

```
lifecycle {
    create_before_destroy = true
 }
```

Terraform – AWS Autoscaling & Loadbalancing

```
resource "aws_alb_listener_rule" "send_all_to_web_servers" {
  listener_arn = aws_alb_listener.http.arn
  priority     = 100

  action {
    type             = "forward"
    target_group_arn = aws_alb_target_group.web_servers.arn
  }

  condition {
    field  = "path-pattern"
    values = ["*"]
  }
}
```

Lab 9 – Scaling & loadbalancing web servers

In this lab we will see how to create the necessary resources to autoscale a group of Web Servers and to load-balance traffic to them

Quiz

What resource is responsible to create the web server instances ?

Why do we specify “*create_before_destroy*” ?

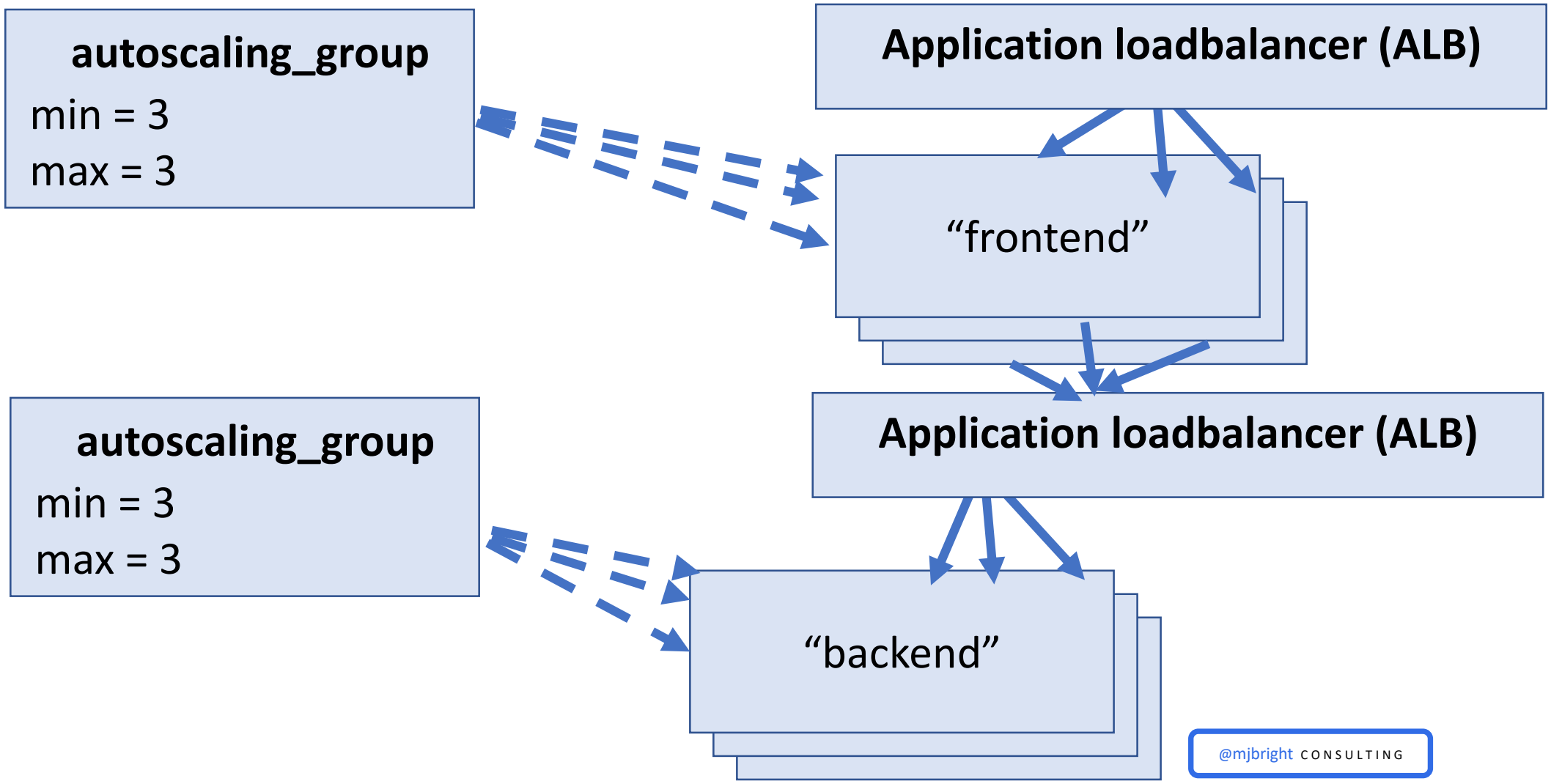
Why do we specify “*depends_on = ["aws_alb_listener.http"]*” ?

How does the ALB TargetGroup verify that web_servers are ready ?

Terraform – AWS Autoscaling & Loadbalancing

We'll implement the scenario as a micro-service module facilitating the implementation of both backend & frontend load-balancing

Terraform – AWS Autoscaling & Loadbalancing



Lab 10 – Using a micro-service architecture

In this lab we will revisit the autoscaling & load-balancing across a group of Web Servers

This time we will refactor our code so that Autoscaling/Loadbalancing is implemented in a "*microservices*" module

This will allow us to autoscale/loadbalance backend and frontend resources independently

Quiz

What are the 3 types of load balancer provided by AWS ?

What differentiates them ? On what basis would you choose ?

Terraform – AWS VPCs

So far we have only used the default VPC – “Virtual Private Cloud”

Each region has a default VPC, but we can create up to 5 custom VPCs per region

Introduction to VPCs
VPC & NAT
EC2 instances in VPC

Terraform – AWS VPCs

A VPC is a logically isolated virtual network where you can launch AWS resources

The virtual network has your choice of the following **AWS VPC components:**

- VPC CIDR Block
- subnet
- Internet gateway
- Route Table for Egress traffic
- Security group for Ingress/Egress traffic (stateful)
- Access Control Lists – firewall

Both IPv4 & IPv6 are available

https://aws.amazon.com/vpc/

Terraform – AWS VPCs

The default VPC may already fit your needs

Reasons to set up a new VPC rather than using the default VPC

- some not-publicly reachable nodes - “default VPC” instances are reachable
- specific size of CIDR block - “default VPC” CIDR 172.31.0.0/16
- different subnet sizes - “default VPC” subnet per az of region CIDR 172.31.0.0/20

https://aws.amazon.com/vpc/

Terraform – AWS VPCs

```
resource "aws_vpc" "vpc" {
    cidr_block = var.cidr_block
    instance_tenancy = "dedicated“
    enable_dns_support = true
    tags = {
        Name = "${var.env}_vpc"
        Env = var.env
    }
}

resource "aws_subnet" "subnet" {
    vpc_id = aws_vpc.vpc.id
    cidr_block = var.subnet
    map_public_ip_on_launch = "true"
}
```

Terraform – AWS VPCs

```
resource "aws_internet_gateway" "gw" {
     vpc_id = aws_vpc.vpc.id
}


resource "aws_default_route_table" "route_table" {
      default_route_table_id =
aws_vpc.vpc.default_route_table_id

      route {
            cidr_block = "0.0.0.0/0"
            gateway_id = aws_internet_gateway.gw.id
      }
}
```

Terraform – AWS VPCs

```
resource "aws_instance" "VM-in-new-vpc" {
    ami             = "ami-221ea342"
    instance_type = "m3.medium“

    subnet_id = var.subnet_id
    vpc_security_group_ids = var.vpc_security_group_ids
}
```

Terraform – AWS EBS

Elastic Block Store (EBS) provides raw block-level storage attachable to Amazon EC2 instances
It is used by Amazon Relational Database Service (RDS)

Amazon EBS provides a range of options for storage performance and cost.

These options are divided into two major categories:

- SSD-backed storage for transactional workloads, such as DBs & boot volumes (IOPS)
- disk-backed storage for throughput intensive workloads, such as MapReduce & log processing (MB/s)

Terraform – AWS EBS

```
resource "aws_ebs_volume" "ebs-volume-1" {
    availability_zone = us-west-1a"
    size = 20
    type = "gp2"
    tags {
        Name = "extra volume data"
    }
}

resource "aws_volume_attachment" "ebs-volume-1-attachment" {
  device_name = "/dev/xvdh"
  volume_id = "${aws_ebs_volume.ebs-volume-1.id}"
  instance_id = "${aws_instance.example.id}"
}
```

Thank you !

 https://linkedin.com/in/mjbright

 @mjbright

 @mjbright

Resources

Resource	URL
Website	https://www.terraform.io https://www.terraform.io/docs
Downloads	https://www.terraform.io/downloads.html https://github.com/hashicorp/terraform/releases
Learn Terraform	https://learn.hashicorp.com/terraform/getting-started/intro https://www.terraform.io/docs/glossary.html
Terraform Registry	https://registry.terraform.io/

Resources

Resource	URL
	https://www.terraform.io/docs/cloud/guides/recommended-practices/index.html
	https://www.terraform-best-practices.com @antonbabenko
	https://github.com/ozbillwang/terraform-best-practices

Resources

Resource	URL
	https://github.com/shuaibiyy/awesome-terraform https://github.com/kapil12345/awesome-terraform

Resources

Resource	URL
Books	
	« Terraform Up & Running », O'Reilly,
	"The Terraform Book", Turnbull Press
	"Terraform in Action", Manning, Scott Winkler

Resources

Resource	URL
	https://github.com/gruntwork-io/terraform-aws-couchbase.git

Thank you !

 https://linkedin.com/in/mjbright

 @mjbright

 @mjbright