Creating Cloud Resources with Terraform

Michael Bright , Terraform & Vault Associate

Webinar Agenda

1. Introduction to Terraform
   - Configurations, Providers
   - Installation, Workflow
   - Variables: Input, Output, Local

2. Control structures
   - Replicas using Count

3.Working with Terraform Registry
   - Data sources
   - Modules

4. Terraform in Production
   - Provisioners
   - Managing State

@mjbright

1. Introduction

Terraform Concepts

HashiCorp Products & Projects

HashiCorp

HashiCorp
Terraform

Terraform CDK
Python/Typescript

+ Terraform Cloud/
Terraform Enterprise
[Sentinel]

Waypoint

BOUNDARY

HashiCorp
Consul

HashiCorp
Vault

HashiCorp
Packer

HashiCorp
Vagrant

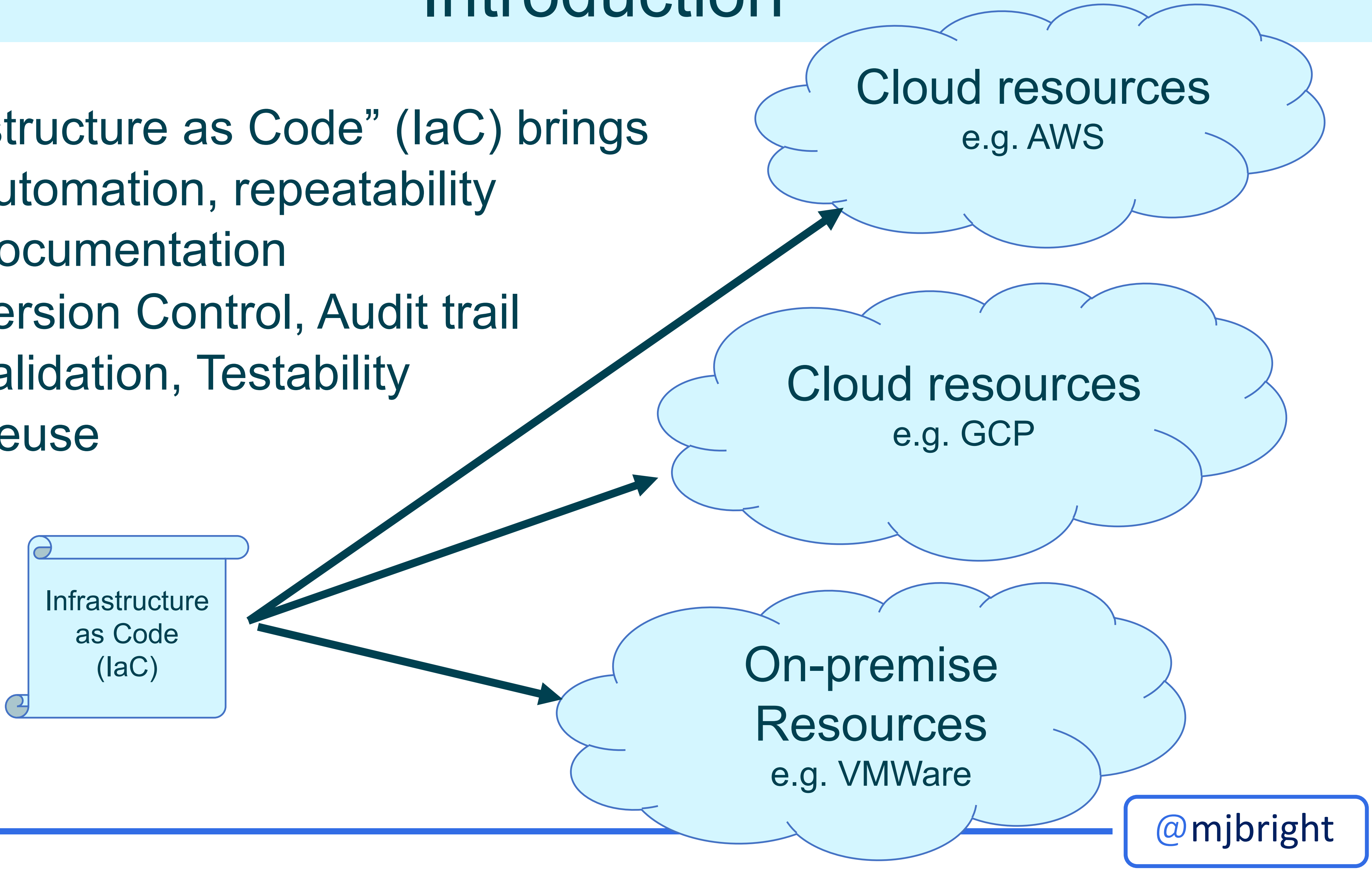
HashiCorp
Nomad

@mjbright

Introduction

"Infrastructure as Code" (IaC) brings

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



Introduction

Infrastructure as Code (IaC) is a descriptive model, where source code describes the infrastructure resources to be created.

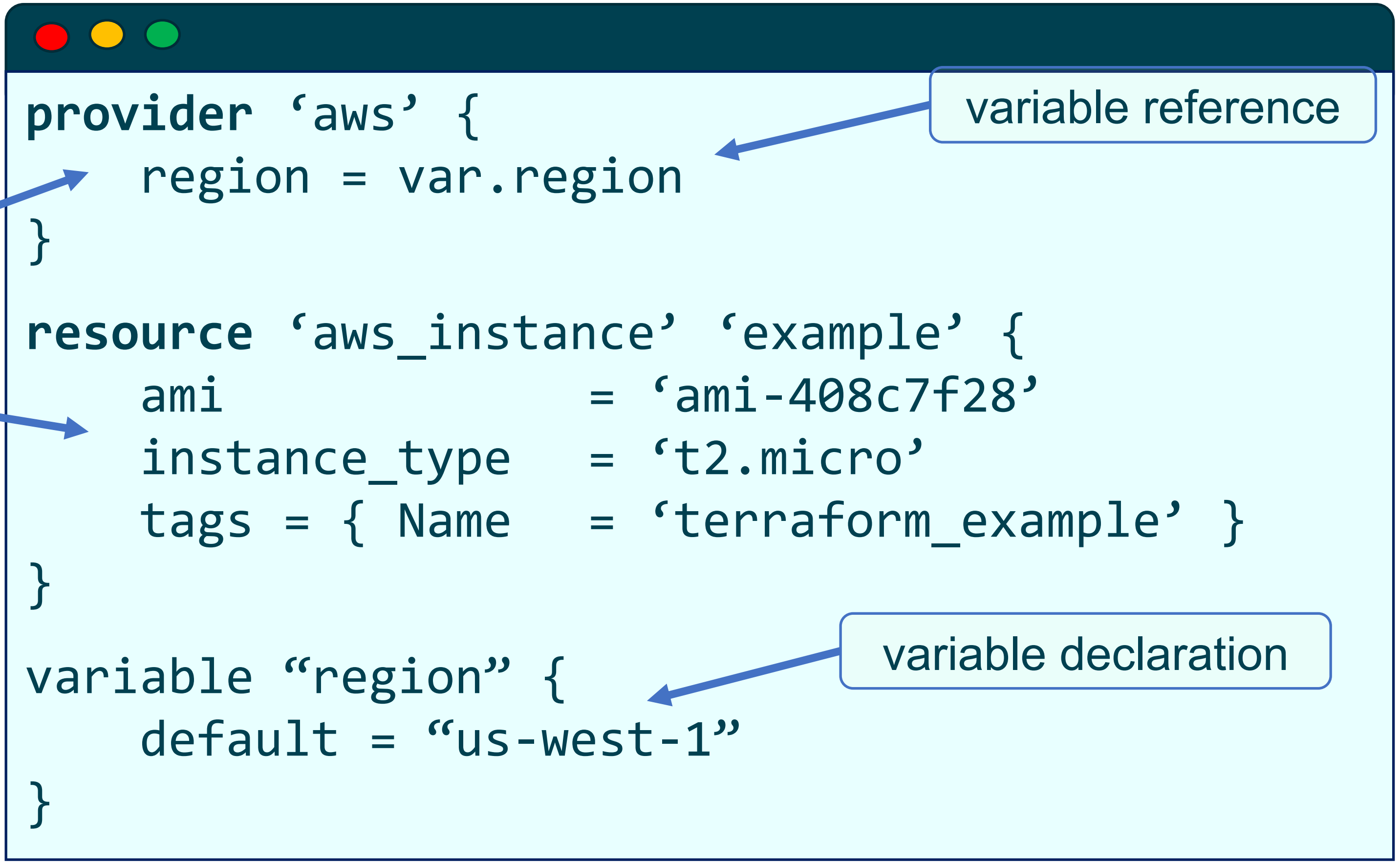
In this model re-applying the same configuration assures the same result, i.e. the same set of resources and state, every time.

Terraform Configurations

Example Terraform Configurations

Configurations use Hashicorp Configuration Language (HCL v2) & specify a provider & resources specific to the provider

attributes



```
provider ‘aws’ {
    region = var.region
}

resource ‘aws_instance’ ‘example’ {
    ami              = ‘ami-408c7f28’
    instance_type    = ‘t2.micro’
    tags = { Name    = ‘terraform_example’ }
}

variable “region” {
    default = “us-west-1”
}
```

variable reference

variable declaration

The Terraform CLI evaluates & applies configurations, using provider plugins which define & manage a set of resource types

JSON format can also be used – files named as .tf.json

@mjbright

(Some) Terraform Providers

Cloud

AWS, Alibaba Cloud, Azure, DigitalOcean, Exoscale, Google Cloud, Heroku, IBM Cloud, Oracle Cloud, OVH, Packet, 1&1, Spotinst, Linode

HashiCorp

Vault, Nomad, Consul

Version Control

GitHub, GitLab, Bitbucket, Azure DevOps

Orchestrators

Kubernetes, Helm, Docker, OpenStack

Databases

PostgreSQL, MySQL, MongoDB Atlas

Hypervisors

VMware NSX-T, vCloud , vSphere, **Proxmox**

Misc

Cobbler, Datadog, DNS, HTTP, Local, TLS

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

What about you ?

A quick anonymous survey

What Platforms do you currently use ?

What is your experience with IaC ?

What Providers would you like to use with Terraform ?

https://forms.gle/A7W1DE5or65fJUH59

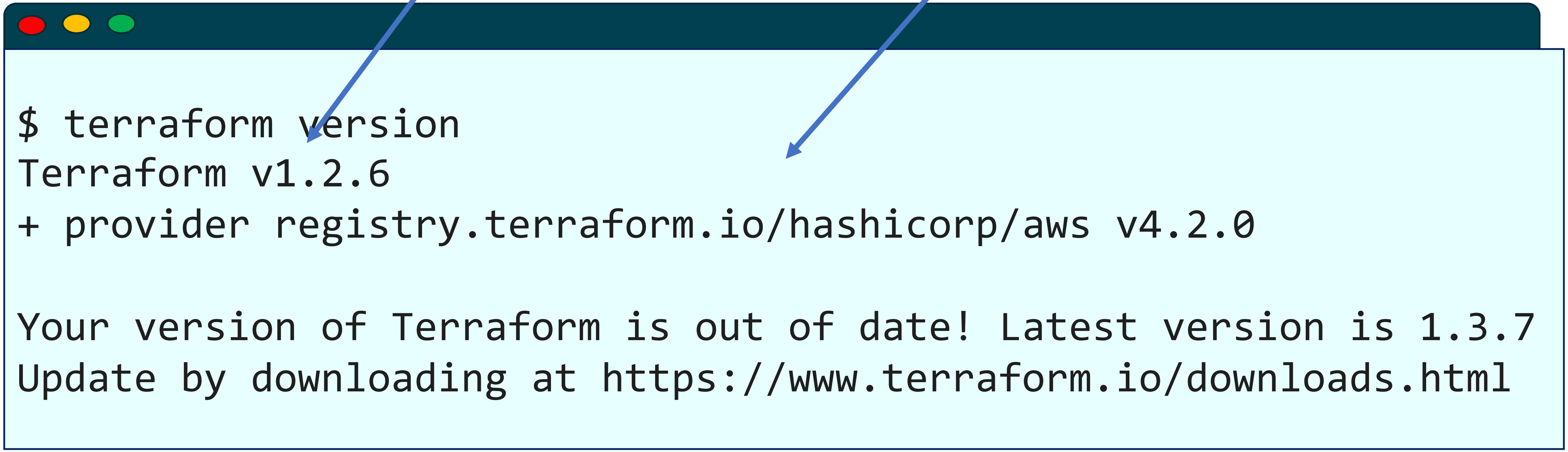
https://www.terraform.io/docs/providers/index.html

Terraform Installation

Terraform - Installation

Terraform version

Provider plugins installed – if present
(installed at ‘*terraform init*’)



```
$ terraform version
Terraform v1.2.6
+ provider registry.terraform.io/hashicorp/aws v4.2.0

Your version of Terraform is out of date! Latest version is 1.3.7
Update by downloading at https://www.terraform.io/downloads.html
```

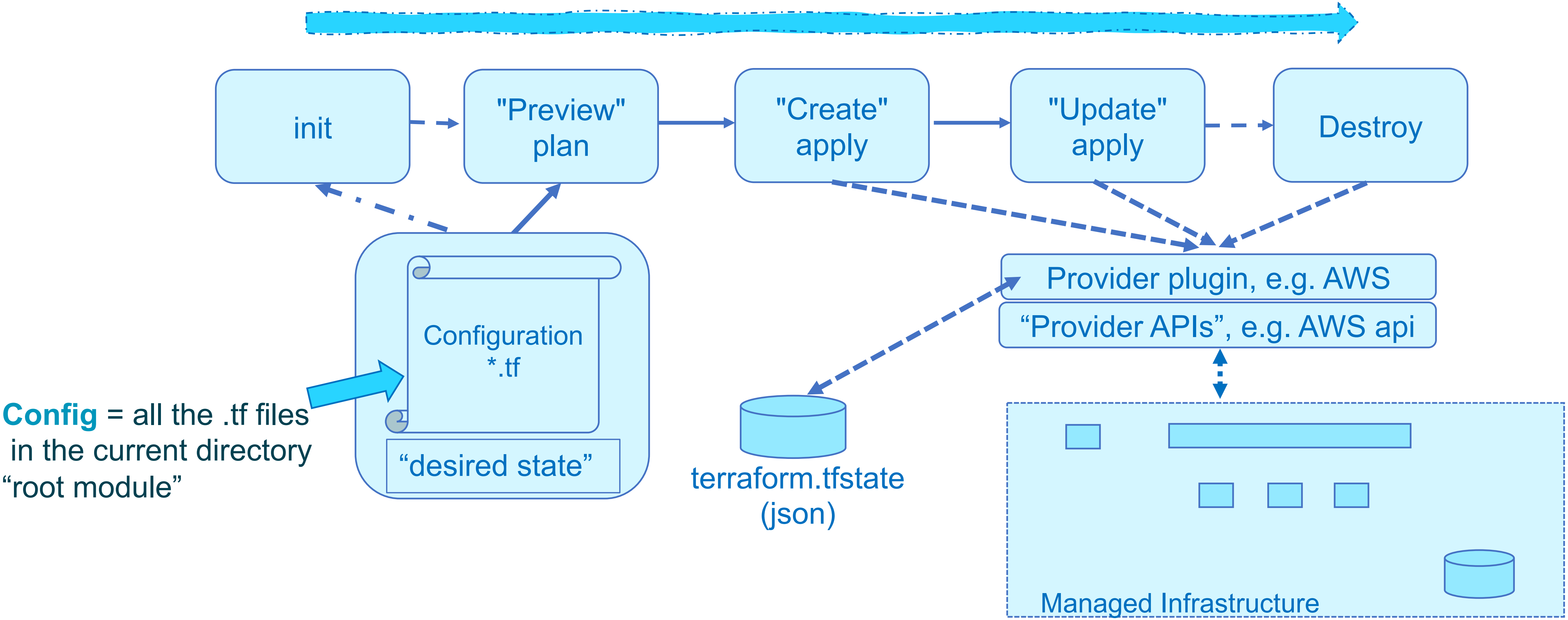
Download latest from: https://www.terraform.io/downloads.html
Older or beta releases from: https://releases.hashicorp.com/terraform/

Terraform Workflow

Planning, applying, updating, destroying

Terraform Workflow

Terraform determines the current state of the resources it is managing



Config = all the .tf files in the current directory
"root module"

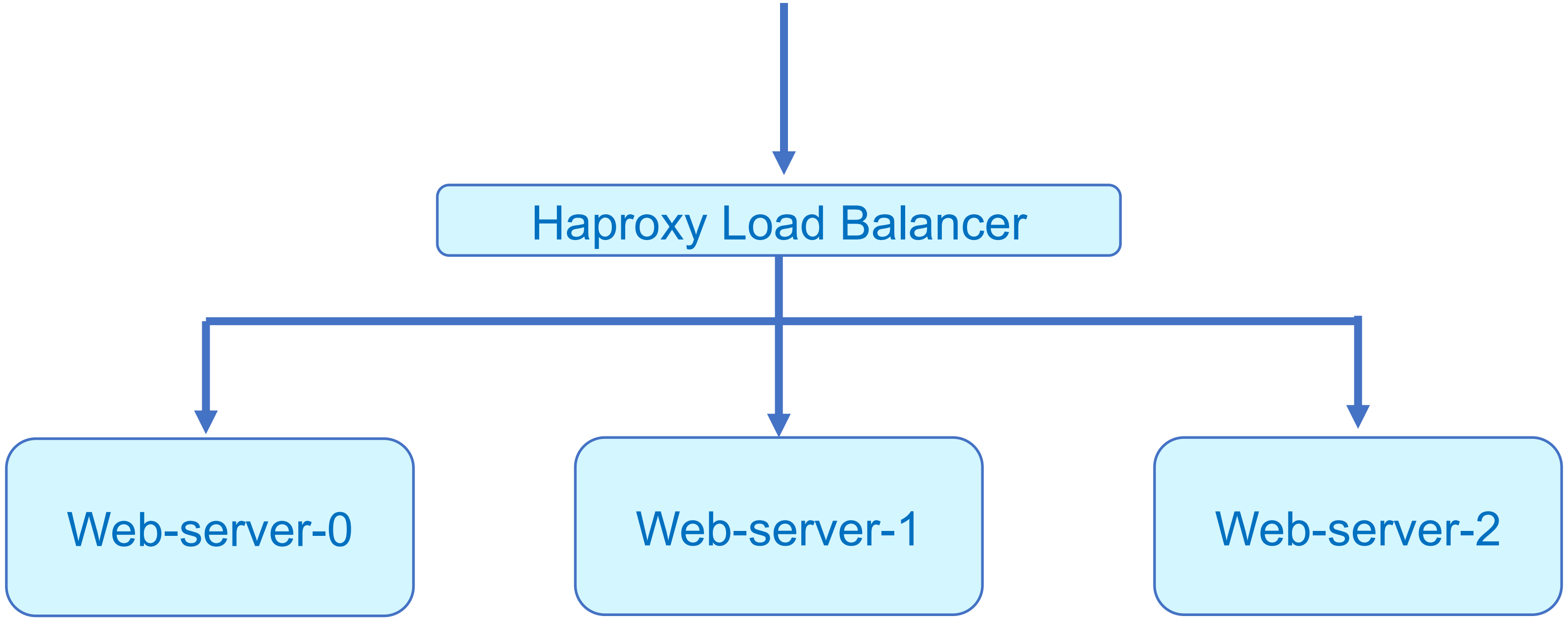
https://www.terraform.io/docs/commands/index.html

@mjbright

What will we do ?

Building up a small 2-tier architecture

Terraform Workflow



Only a simple architecture right ?

But how about if we can fire it up with 2 commands, in less than 10 seconds ?

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

HCL Configs, variables & Control Structures

Parameterizing Terraform configurations through variables

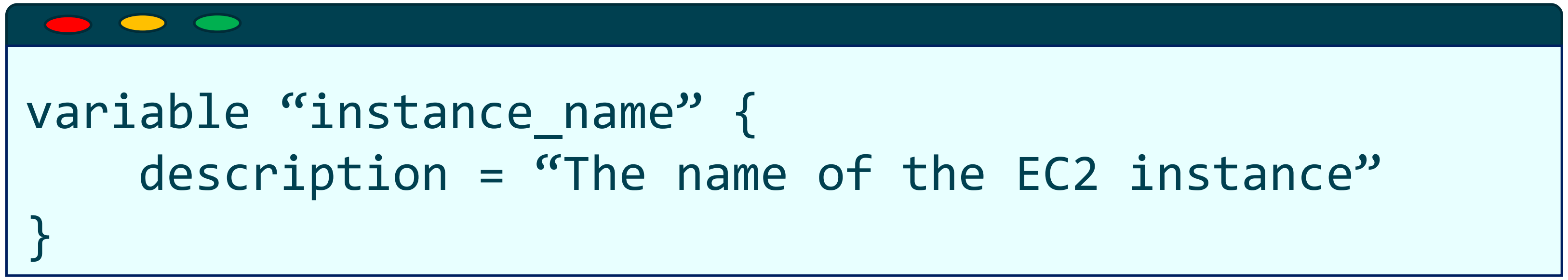
Terraform Configurations

@mjbright

Parameterizing Terraform Configurations

We can parameterize our templates using *variables.*

Description, default and type attributes are optional.



```
variable “instance_name” {
    description = “The name of the EC2 instance”
}
```

It is best practice is to

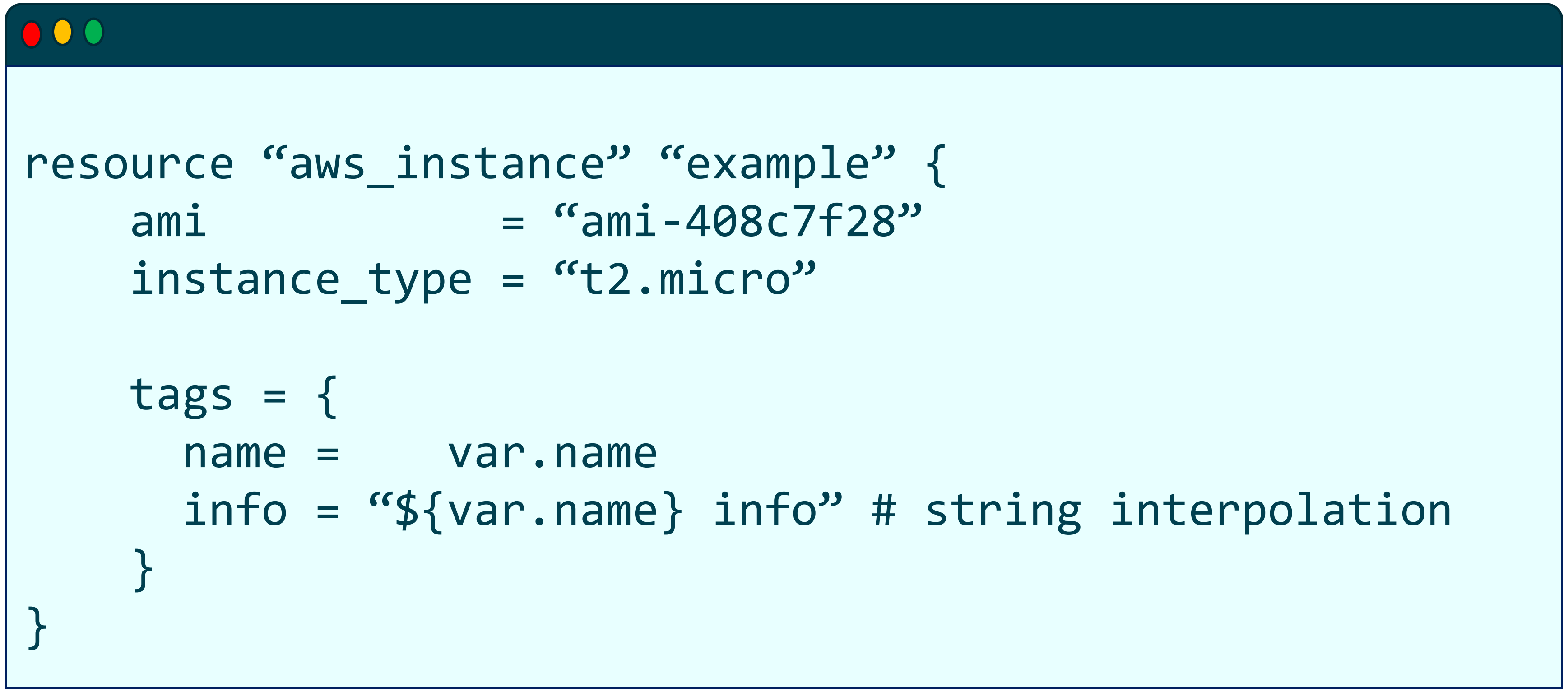
Define the variables (as above) in a **variables.tf** file

Specify the default values in a **terraform.tfvars** file

```
instance_name = “resource-value”
```

Parameterizing Terraform Configurations

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



```
resource “aws_instance” “example” {
    ami           = “ami-408c7f28”
    instance_type = “t2.micro”

    tags = {
      name =    var.name
      info = “${var.name} info” # string interpolation
    }
}
```

Outputting values

Outputting values – single instance

We can specify output values in our configuration templates:

```
output "public_ip"  { value = aws_instance.ex.public_ip     }
output "public_dns" { value = aws_instance. ex.public_dns }
output “hosts"      { value = "IP: ${aws_instance. ex.public_ip} Host:
                               ${aws_instance.example.public_dns}" }
```

which will be shown at the end of the apply

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

Outputs:
public_dns = ec2-54-183-131-114.us-west-1.compute.amazonaws.com
public_ip  = 54.183.131.114
Hosts      = IP: 54.183.131.114 Host: ec2-54-183-131-114.us-west-1.compute.amazonaws.com
```

and thereafter can be viewed with commands
‘*terraform show*’ or ‘*terraform output*’

Terraform Types

@mjbright

Terraform Types

Types

Terraform supports different types of variables

- Number, String, Boolean
- Lists, Tuples, Sets, Objects
- Maps
- Any to wildcard collections e.g. map(any), list(any)
- null (for omitted values)

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

Local Variables

@mjbright

Locals

Locals are pre-computed values which can be reused elsewhere in the same module

Locals simplify configurations by declaring formulas only once, improving readability if meaningful names are used

```
locals {
    info_val = "just a reusable string“
    size     = 10
    area     = local.size * local.size * 3.1415926835
}
```

Locals apply across all your configuration files of the current module

```
output test_local  { value = local.info_val }
output circle_area { value = local.area     }
output approx_area { value = “approx. size=${floor(local.area) cm^2"}
```

Terraform Control Structures

HashiCorp
Terraform

Abstractions for (looping) creating multiple resources

Terraform Control Structures

Terraform provides a bewildering array of control structures

- Count loops
- Ternary operator (if)
- For_each loops
- “for in” loops
- Dynamic blocks
- String templates

Count Loops

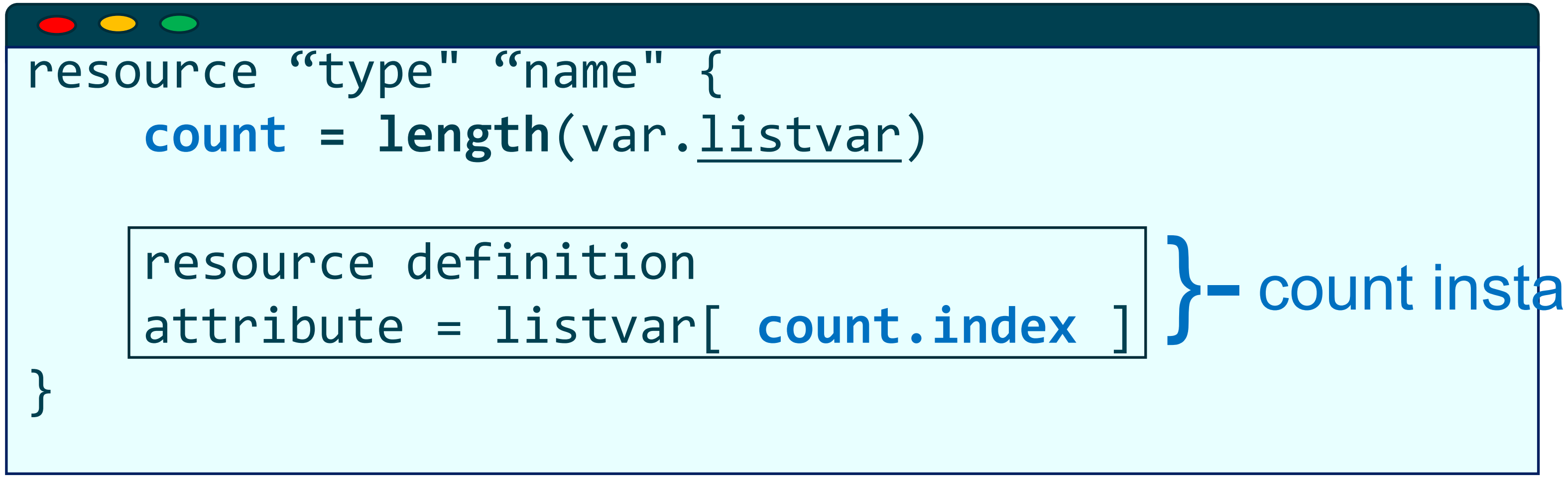
Terraform Control Structures: “count” loops

Use of the *count* attribute causes creation of multiple resource instances:

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

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

Note: elements begin with index zero, not one!



```
resource “type" “name" {
    count = length(var.listvar)

    resource definition
    attribute = listvar[ count.index ]
}
```

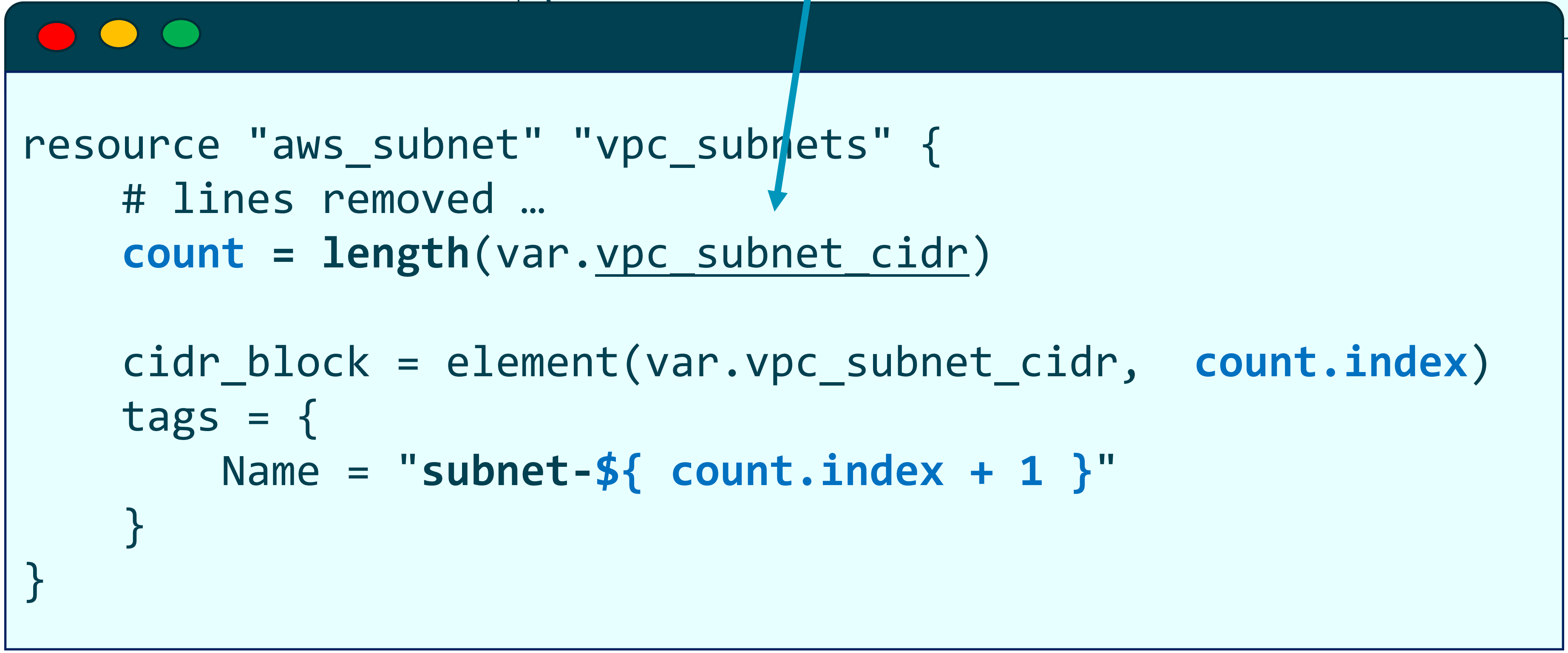
count instances created

The created resources can be referenced as type.name[0], [1] etc …

Terraform Control Structures: “count” loops

Example: To create several aws_subnet instances **(using count)**

```
variable “vpc_subnet_cidr" {
    default = [ “192.168.0.0/24”, “192.168.1.0/24” ]
}
```



```
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 }"
    }
}
```

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

https://www.terraform.io/docs/configuration/resources.html#count-multiple-resource-instances-by-count

Webinar Agenda

1. Introduction to Terraform
   - Configurations, Providers
   - Installation, Workflow
   - Variables: Input, Output, Local

2. Control structures
   - Replicas using Count

3.Working with Terraform Registry
   - Data sources
   - Modules

4. Terraform in Production
   - Provisioners
   - Managing State

Data Sources

HashiCorp
Terraform

Querying the Provider for information

Terraform Data Sources

Filtering the data

Example: Get the latest Ubuntu 20.04 LTS (for this region)

```
# Get latest Ubuntu Fossa 20.04 AMI
data "aws_ami" "ubuntu-linux-2004" {
  most_recent = true
  owners      = ["099720109477"] # Canonical
  filter {
    name   = "name"
    values = ["ubuntu/images/hvm-ssd/ubuntu-fossa-20.04-amd64-server-*"]
  }
  filter {
    name   = "virtualization-type"
    values = ["hvm"]
  }
}
```

Terraform Data Sources

Example: Using the aws_ami data source

```
resource “aws_instance" "server" {
    name     = "tf-data-source-example“

    key_pair = "terraform“
    ami      = data.aws_ami.ubuntu-linux-2004.id
}
```

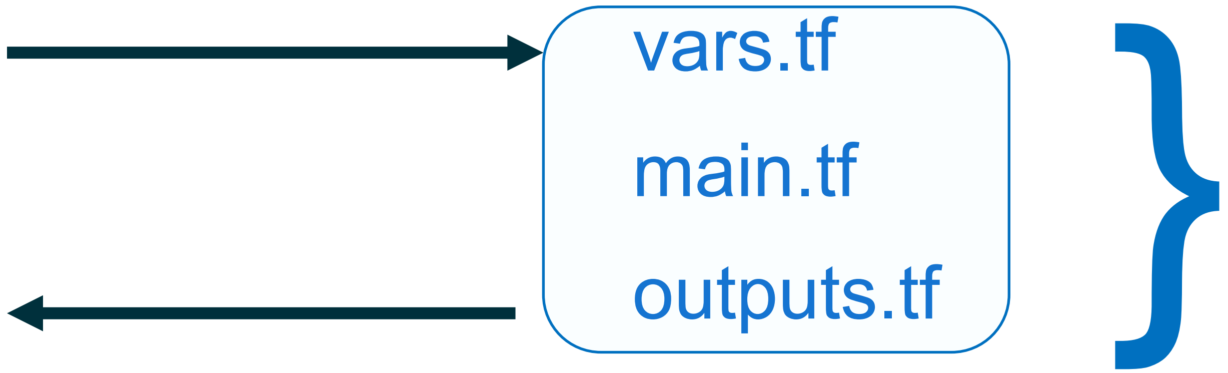
Terraform Modules

Modules similar to functions in other languages

Terraform Modules

Local source Modules

By convention we define 3 specific terraform files in a module



vars.tf

main.tf

outputs.tf

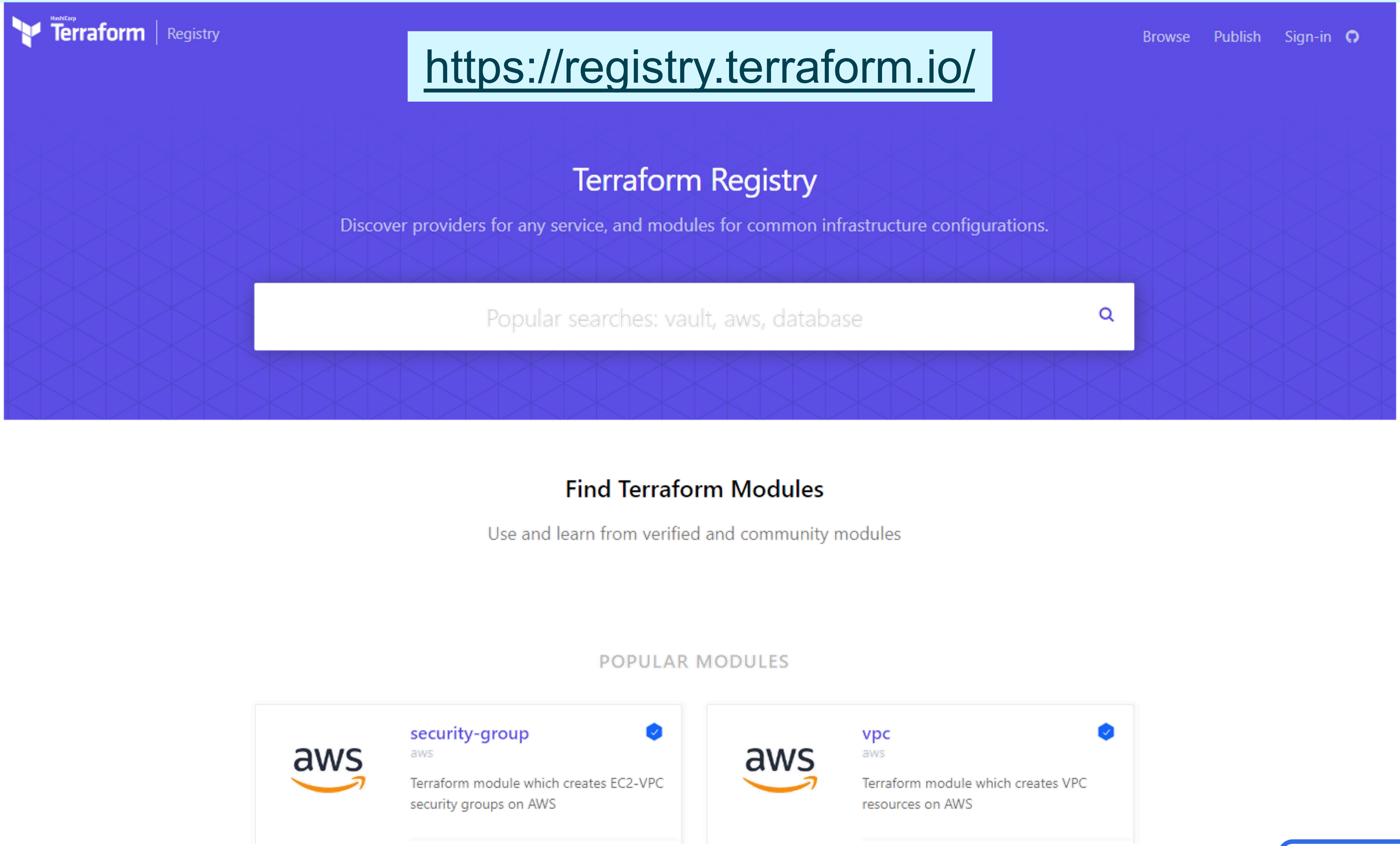
Module is “callable” with input/output variables

A module may also define its’ own local variables

But this is mere convention, we can call these files as we wish.

As modules become more complex it makes sense to split the configuration accordingly

Terraform Modules - Terraform Registry



https://registry.terraform.io/

Terraform Registry

Discover providers for any service, and modules for common infrastructure configurations.

Popular searches: vault, aws, database

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

Terraform in Production

HashiCorp
Terraform

Provisioners
Importing “foreign” resources
Managing State
Delegating state to remote storage - teams

Provisioners

Not the same as Providers !

Terraform Provisioners & user_data

Provisioners & user_data

User_data
A common mechanism used to provision cloud resources is **user_data** which can be used to execute shell commands using Linux **cloud-init** capabilities.

Provisioners
Another mechanism specific to Terraform is **Provisioners**:
Local-exec: Execute code on local machine, which invokes Terraform
Remote-exec: Execute code on remote machine managed by Terraform
File: Transfer files to the remote machine

“*Ultimate Terraform*” trainingAgenda - *

(for 4 half-day training, Apr 24th to 27th)

1. Introduction to Terraform
   - Configurations, Providers
   - Installation, Workflow
   - Variables, types

2. Control structures
   - Replicas, templates, …
   - Resource dependencies
   - HCL functions, Debugging

3.Working with Terraform Registry
   - Providers & Data sources
   - Using & Writing Modules

4. Managing State
   - Provisioners
   - Managing Remote State
   - Importing Foreign Resources

5. Terraform in Production
   - Best Practices
   - 3rd-party tools
   - Terraform Cloud & Enterprise

@mjbright

* - subject to change

Thank you !

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