# Terraform with AWS

**Hosted by Marcus Ross for CGI**

**Version:** 1.0.3

# Course Details

We have two sessions (days) to cover the agenda.

- Session 1 @ 9AM-4PM Terraform Fundamentals and AWS Core Services
- Session 2 @ 9AM-4PM Terraform Advanded and more AWS Services

# Goals of the Course

- use Terrafrom to deploy some Infrastructure
- Hands-On Time
- getting a refresh on AWS services

# Agenda Day 1

- Install Terraform and AWS-Connection
- Why IaC / Terraform in general
- First Example (EC2-HelloWorld)
- Syntax Essentials and Best Practices
  - Core Developer Loop
  - Variables
  - Essential Functions
  - File-Layout
  - Dependencies
- Terraform with AWS
  - Create EC2-Instances
  - Create S3 Buckets
  - Create a VPC

# Agenda Day 2

- Host WebSites with S3-Buckets
- Create EC2-Cluster and a Load-Balancer
- S3
  - Host Remote State with S3-Buckets
- Route 53 to create a friendly DNS Entry
- IAM Users and Policies
- RDS
  - Deploy a PostgresDB
  - Deploy a Aurora-ServerlessDB
- EKS
  - Deploy an EKS-Cluster
  - Deploy an NGINX-POD
- Wordpress End-2-End Example

# Manual Configuration Challenges

- Creating and configuring services is often done manually
- Documentation
- Reliability
- Reproducibility
    - Dev
    - Test
    - Prod

# What is Infrastructure as Code?

> “Infrastructure as Code is the process of managing and provisioning computer data centers through machine-readable definition files, rather than physical hardware configuration or interactive configuration tools”

Source: Wikipedia

# Is Terraform the only how does IaC?





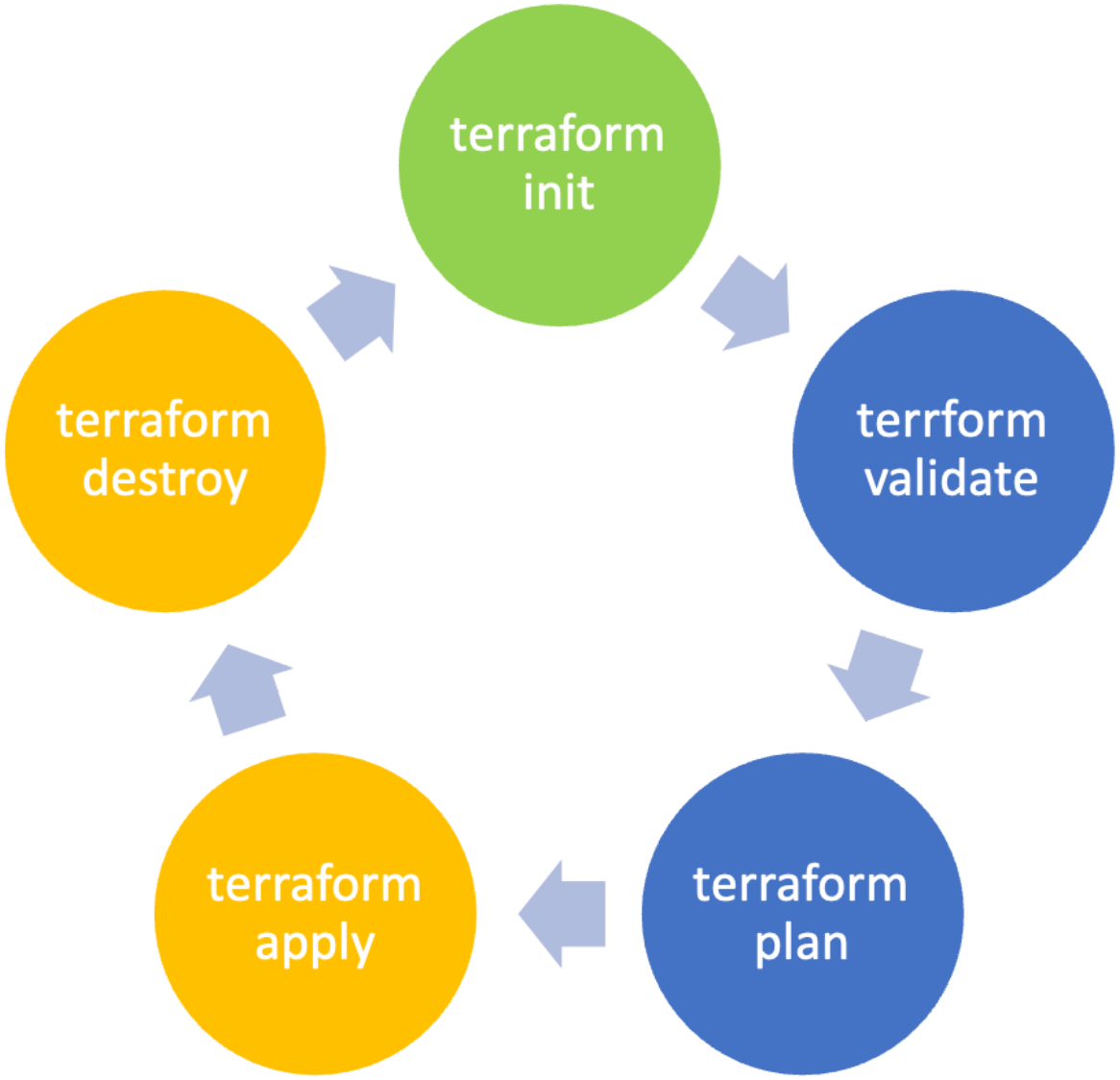
# Terraform – Template Example

This template creates a single EC2 instance in AWS

```
provider "aws" {
  region = "us-east-2"
}

resource "aws_instance" "example-instance" {
  ami           = "ami-0e5b6b6a9f3db6db8"
  instance_type = "t2.micro"
}
```

# Terraform – Core Loop



# Terraform – Key capabilities

- Terraform is a tool for provisioning infrastructure
- supports many providers (cloud agnostic)
- many resources for each provider
- define resources as code in terraform templates

PRODUCTS & TECHNOLOGY

# Announcing HashiCorp Terraform 1.0 General Availability

Terraform 1.0 — now generally available — marks a major milestone for interoperability, ease of upgrades, and maintenance for your automation workflows.

JUN 08 2021 | KYLE RUDDY

# Terraform 1.0 - What are the beneifts?

- Extended Maintenance Periods
  (1.x releases have 18 month maintenance period)
- More mature and stable
  (essentially a 0.15 super-service pack)
- Terraform state is cross-compatible between versions
  (0.14.x, 0.15.x, and 1.0.x.)

# LAB

## Setup & "Hello Infra"

- Install and Setup Terraform
- create IAM User in AWS (AWS-CLI/Console)
- Initialize the aws-Provider
- define EC2-Instance and apply

# create a very risky simple ec2-instance (main.tf)

```
provider "aws" {
  region = "us-west-2"
}

resource "aws_instance" "app_server" {
  ami = "ami-830c94e3"
  instance_type = "t2.micro"

  tags = {
    Name = "ExampleAppServerInstance"
  }
}
```

Why is this a weak example in the sense of IaC and **not** AWS perspective?

# Terraform Version constraints

specify a range of acceptable versions (">= 1.2.0, < 2.0.0")

```
terraform {
  required_providers {
    aws = {
      source  = "hashicorp/aws"
      version = "~> 3.27"
    }
  }
  required_version = "≥ 1.0"
}
```

# a better approach for a simple ec2-instance (main.tf)

```
terraform {
  required_providers {
    aws = {
      source = "hashicorp/aws"
      version = "~> 3.27"
    }
  }
  required_version = ">= 1.0"
}

provider "aws" {
  region = "us-west-2"
}

resource "aws_instance" "app_server" {
  ami = "ami-830c94e3"
  instance_type = "t2.micro"

  tags = {
    Name = "ExampleAppServerInstance"
  }
}
```

# Working with a state file

- Terraform saves everything about the instance to special file(s)
- Same directory as template file with `.tfstate` extension
  - terraform.tfstate
  - terraform.tfstate.backup
- The statefile should **not** be **committed** into version control
- This can be a problem on multi-developer env´s (more about that tomorrow)

# LAB

## manage drift with Terraform

- check `terraform.tfstate`
- change Instance-Type to `t3.micro`
- add a `costcenter=42` Tag
- `$ terraform apply` changes
- check the statefile again
- change costcenter via Dashboard
- `terraform plan` and check if TF can manage this drift

# Terraform *Standard* Filelayout

| File / Folder | Purpose |
|---|---|
| main.tf | Terraform Config and Constraints |
| output.tf | Output like IPs, Addresses, etc |
| provider.tf | Provider-Specific (Cred.) |
| ressources.tf | for small projects |
| variables.tf | place for specifying variables |
| README.md | Documentation |
| env | folder place for tfvar-files |

# LAB

## please refactor your code

- add `provider.tf` and refactor
- add `ressource.tf` and refactor
- create empty file `variables.tf`
- create empty file `output.tf`
- create an `env` folder

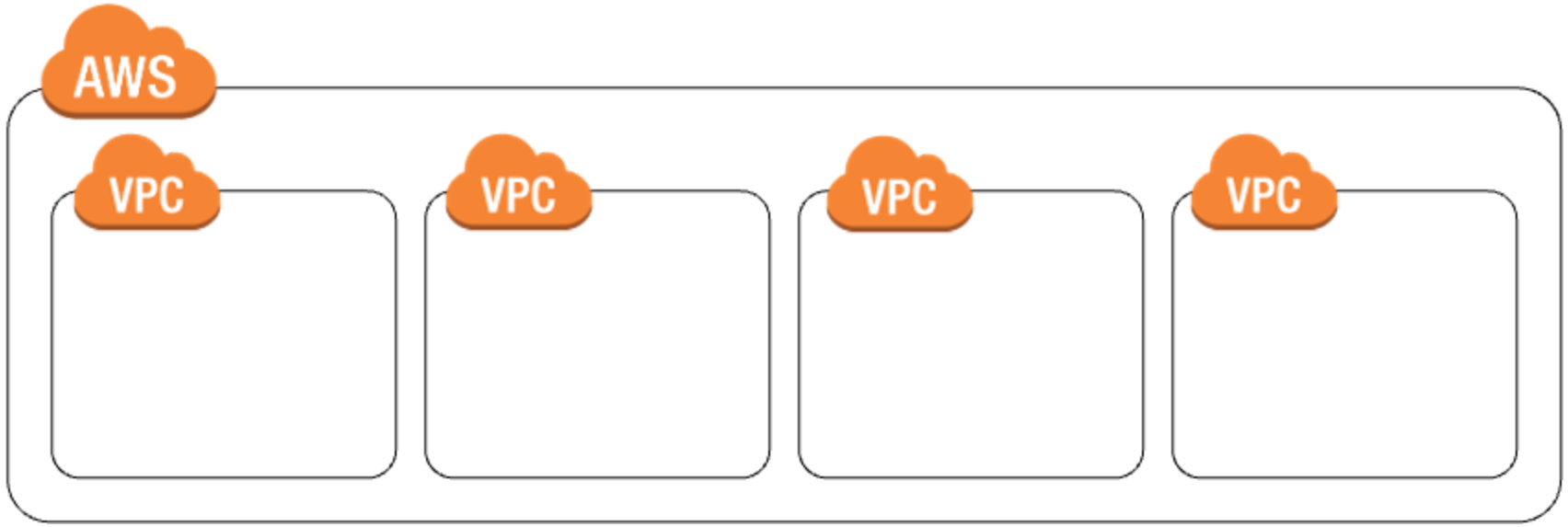

# Virtual Privat Cloud

- A VPC is a virtual network dedicated to your AWS account
- Requires an IPv4 address space and optionally IPv6 address ranges
- Enables you to create specific CIDR ranges for your resources to occupy
- Provides strict access rules for inbound and outbound traffic.

# Multi-VPC per Account

**Best suited for:**

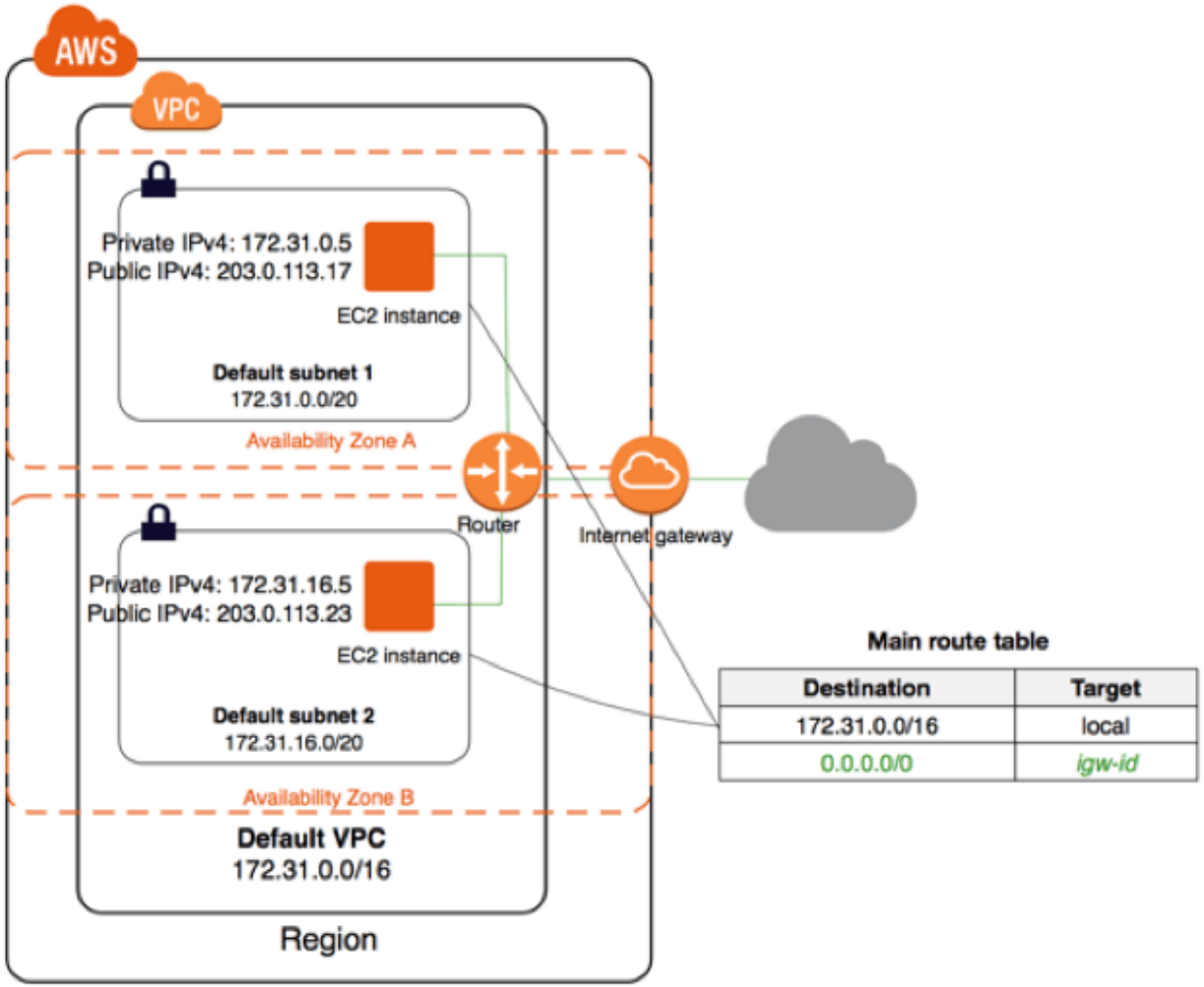
- single team or single organizations, such as managed service providers
- limited teams, which makes it easier to maintain standards and manage access



# Components of a VPC

- VPC CIDR Block
- Subnet
- Gateways
- Route Table
- Network Access Control Lists
- Security Groups

Source from Rackspace Blog



# LAB

## Setup your VPC (hard-way)

- create a VPC in eu-central-1
- create a public Subnet for one AZ
- create an Internet-GW
- create a Route Table

# getting replicas with count

This template creates 4 single EC2 instance in AWS

```
resource "aws_instance" "app_server" {
  count         = 4
  ami           = "ami-830c94e3"
  instance_type = "t2.micro"

  tags = {
    Name = "App Server ${count.index+1}"
  }
}
```

# Welcome to the World of Variables

# Variables - simple types

There are simple types of variables you can set:

- string
- number
- bool

### define a Variable (variables.tf)

```
variable "app_server_instance_type" {
  type        = string
  default     = "t2.micro"
  description = "The aws instance-type"
}
```

## use Variables in HCL (main.tf)

```
resource "aws_instance" "app_server" {
  count         = var.app_server_count
  ami           = var.ami_id
  instance_type = var.app_server_instance_type

  tags = {
    Name = "App Server ${count.index+1}"
  }
}
```

# Custom validation with Rules I (variables.tf)

using a validation block nested within the variable block

```
variable "image_id" {
  type        = string
  description = "The id of the machine image (AMI) to use for the server."

  validation {
    condition     = length(var.image_id) > 4 && substr(var.image_id, 0, 4) == "ami-"
    error_message = "The image_id value must be a valid AMI id, starting with \"ami-\"."
  }
}
```

# Custom validation with Rules II (variables.tf)

you can even use regex for this

```
variable "image_id" {
  type        = string
  description = "The id of the machine image (AMI) to use for the server."

  validation {
    # regex(...) fails if it cannot find a match
    condition     = can(regex("^ami-", var.image_id))
    error_message = "The image_id value must be a valid AMI id, starting with \"ami-\"."
  }
}
```

# LAB

## use Variables & Functions I

- create variable `node_count`
- create variable `ami_id`
- create varible `instance_type`
- create 3 replicas of your EC2

# Variables - Type Map

A Map is a lookup table, where you specify multiple keys with different values

```
# define a map of images
variable "images" {
  type = "map"

  default = {
    eu-central-1 = "image-1234"
    us-west-1    = "image-4567"
  }
}

# getting the value for region eu-central-1
image_id = var.images["eu-central-1"]

# getting the correct value via a lookup
image_id = lookup(var.images, var.region)
```

# Variables - Type List

A list value is an ordered sequence of strings indexed by integers starting with zero.

```
# define a map of images
variable "user_names" {
  type = "list(String)"
  default = ["Admin", "Jane", "Dane"]
}

# getting the value for the first entry
user = var.user_names[0]

# loop through in a ressource
count = length(var.user_names)
user  = var.user_names[count.index]
```

# LAB

## use Variables & Functions II

- create variable `region` and set default to 'eu-central-1'
- change variable type `ami_id` to `map`

# Variable Files

place Terraform variables in a special file ( `*.tfvars` ) and load them with the Terraform command:

```
foo = "bar"

somelist = ["one","two"]

somemap = {
  foo = "bar"
  bax = "qux"
}
```

`$ terraform apply -var-file=env/development.tfvars`

# LAB

## use Variable-Files

- create `production.tfvar`
  - `region` to 'eu-central-1'
  - `instance_type` to t3.micro
  - `node_count` to 2
- create `development.tfvar`
  - `region` to 'eu-west-1'
  - `instance_type` to t2.micro
  - `node_count` to 1

# using environment variables

you can also supply values to your variables by using environment variables

Terraform will automatically read all environment variables with the `TF*VAR*` prefix

**Example variable.tf**

```
variable db_password {
  type = string
}
```

set the value (Linux)

```
export TF_VAR_db_password=Secret123
```

set the value (PowerShell)

```
$env:TF_VAR_db_password=Secret123
```

# Variable Definition Precedence

Terraform loads variables in the following order (later sources taking precedence over earlier ones):

1. Environment variables
2. The terraform.tfvars file, if present.
3. The terraform.tfvars.json file, if present.
4. Any _.auto.tfvars or _.auto.tfvars.json files, processed in lexical order of their filenames.
5. Any -var and -var-file options on the command line, in the order they are provided. (This includes variables set by a Terraform Cloud workspace.)

# Install a Webserver

Usually we can use SSH Access to install manually or use something like Ansible. Here we will use the startup-hook `user_data` from an EC2-Ressource.

```
user_data = << EOF
  #!/bin/bash
  sudo apt-get update
  sudo apt-get install -y apache2
  sudo systemctl start apache2
  sudo systemctl enable apache2
  echo "<h1>Deployed via Terraform</h1>" > /var/www/html/index.html
EOF
```

# file()-Function

we can use the file-function to dynamically load **local** files during deployment:

### Example install_webserver.sh

```
#!/bin/bash
yum install httpd -y
/sbin/chkconfig --levels 235 httpd on
service httpd start
instanceId=$(curl http://169.254.169.254/latest/meta-data/instance-id)
region=$(curl http://169.254.169.254/latest/meta-data/placement/region)
echo "<h1>$instanceId from $region</h1>" > /var/www/html/index.html
```

### Example main.tf in ressource ec2-instance

```
user_data = "${file("install_webserver.sh")}"
```

# LAB

## install a webserver at startup

- create an install script for Apache
- add a user-data section to ec2
- use file-function to load the script
- check the webserver on the host

  `curl localhost`

# Create Security Groups and Rule Objects

```
resource "aws_security_group" "web_access" {
  name        = "web_access"
  description = "Allow port 80 access from outside world"
}

resource "aws_security_group_rule" "allow_webserver_access" {
  type              = "ingress"
  from_port         = 80
  to_port           = 80
  protocol          = "tcp"
  cidr_blocks       = ["0.0.0.0/0"]
  security_group_id = aws_security_group.webdata.id
}
```

# Use Security Groups

```
resource "aws_security_group" "ssh_access" {
  name        = "web_security_group"
  description = "Terraform web security group"
  vpc_id      = vpc-47111266642

  egress {
    from_port   = 0
    to_port     = 0
    protocol    = "-1"
    cidr_blocks = ["0.0.0.0/0"]
  }

  ingress {
    from_port   = 22
    to_port     = 22
    protocol    = "tcp"
    cidr_blocks = ["0.0.0.0/0"]
  }
}
```

# LAB

## security a.k.a. SG

- create a security-group "webserver-access"
- add ingress-rule for port 22 and port 80 open to the world
- add egress-rule for everything open to the world (if necessary)

# dynamic Blocks in Terraform

Within top-level block constructs like resources, expressions can usually be used only when assigning a value to an argument using the `name = expression` form. This covers many uses, but some resource types include repeatable
**nested blocks** in their arguments, which typically represent separate objects that are related to (or embedded within) the containing object:

# Create Security Groups with dynamic blocks

```
locals {
  ports = [80, 443, 22]
}

resource "aws_security_group" "dynamic-demo" {
  name        = "demo-sg-dynamic"
  description = "Dynamic Blocks for Ingress"

  dynamic "ingress" {
    for_each = local.ports
    content {
      description = "description ${ingress.key}"
      from_port   = ingress.value
      to_port     = ingress.value
      protocol    = "tcp"
      cidr_blocks = ["0.0.0.0/0"]
    }
  }
}
```

# Create Security Groups with dynamic blocks and MAPs

```
locals {
  map = {
    "description 0" = {
      port = 80,
      cidr_blocks = ["0.0.0.0/0"],
    }
    "description 1" = {
      port = 22,
      cidr_blocks = ["10.0.0.0/16"],
    }
  }
}
resource "aws_security_group" "map" {
  name        = "demo-map"
  description = "demo-map"

  dynamic "ingress" {
    for_each = local.map
    content {
      description = ingress.key # IE: "description 0"
      from_port   = ingress.value.port
      to_port     = ingress.value.port
      protocol    = "tcp"
      cidr_blocks = ingress.value.cidr_blocks
    }
  }
}
```

# LAB

## using dynamic blocks:

- refactor the security-group "webserver-access" to use dynamic blocks
- add the following ingress-rules

| Port | CIDR-Block | Description |
|---|---|---|
| 22 | only within VPC | ssh access |
| 80 | open to the world | web access |
| 443 | open to the world | tls web access |

# Understand Dependency Graphs

Terraform doesn't simply build resources and write configuration into a state file

Internally, it also manages a dependency graph of all the resources

For dependencies, a directed graph is necessary

# Dependency Graphs in Terraform

- Terraform uses dependency graphs to determine the build and deletion order of resources
- Three different types of nodes in a Terraform graph:
  - Resource node
  - Provider configuration node
  - Resource meta-node

# Controlling dependencies with depends_on

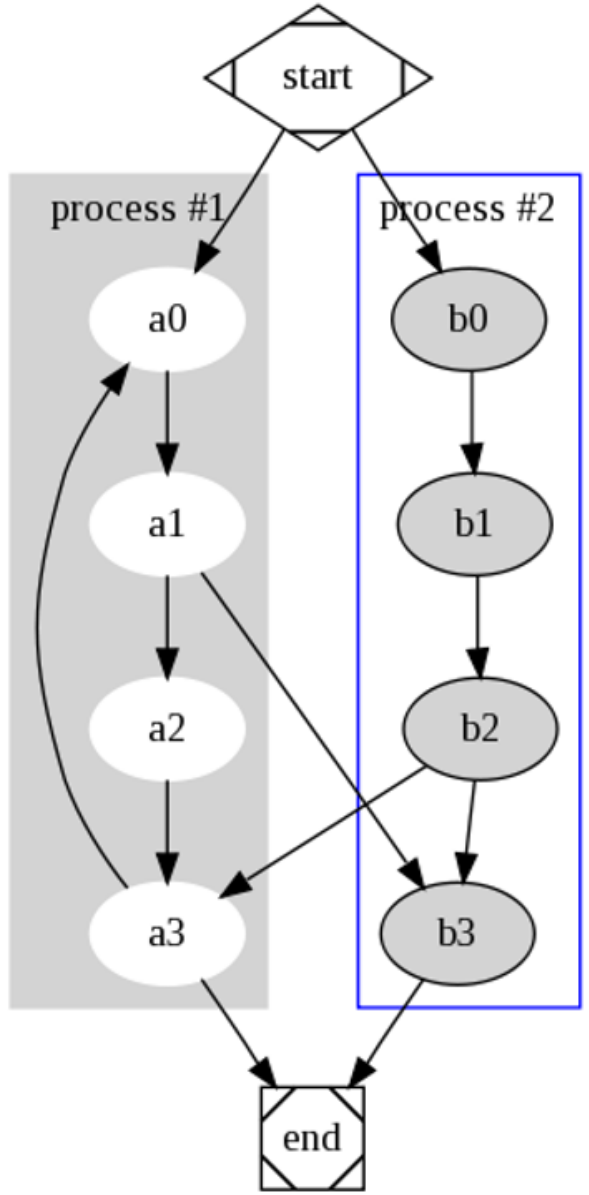
- Usually, Terraform will resolve dependencies automatically
- Sometimes built-in dependency resolution leads to to unwanted behavior
- Enforce dependencies: `depends_on`
  - Accepts a list of resources that this resource depends on
  - Resource won't be created until the ones listed inside this parameter are created

```
resource "aws_instance" "app_server" {
  ami              = var.ami_id
  instance_type    = var.instance_type

  depends_on = [
    aws_instance.db_server
  ]
}
```

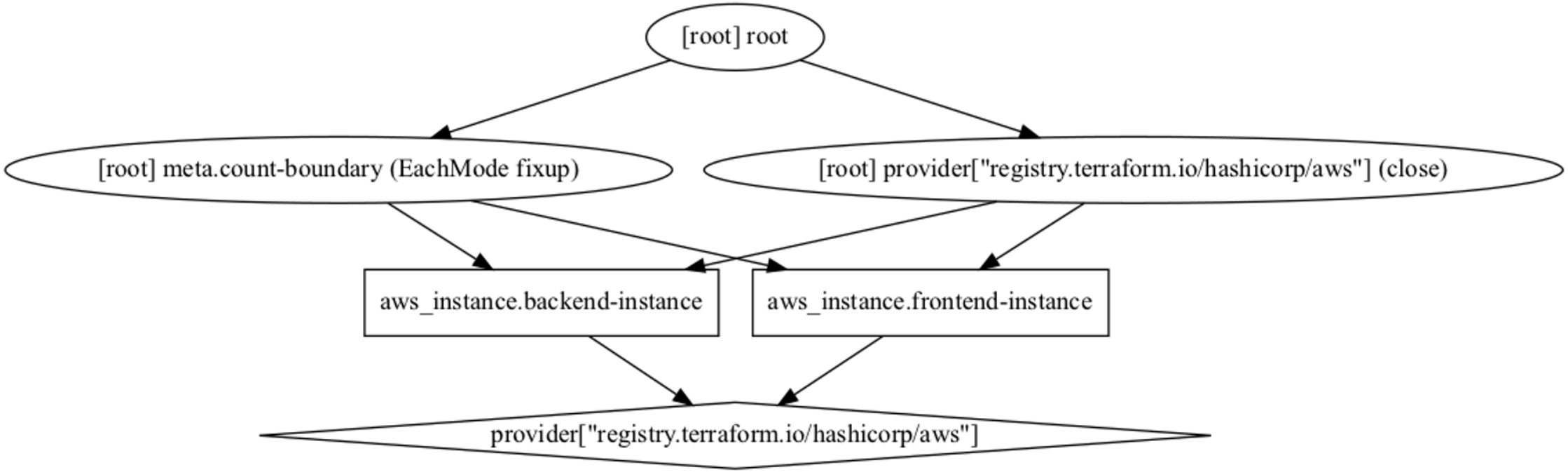
# Terraform Graph Visualization

- Dot-Files are Text-Based
- Use Graphviz to visualize
- Possible output
  - Several pixel images
  - SVG
  - PDF
  - Postscript



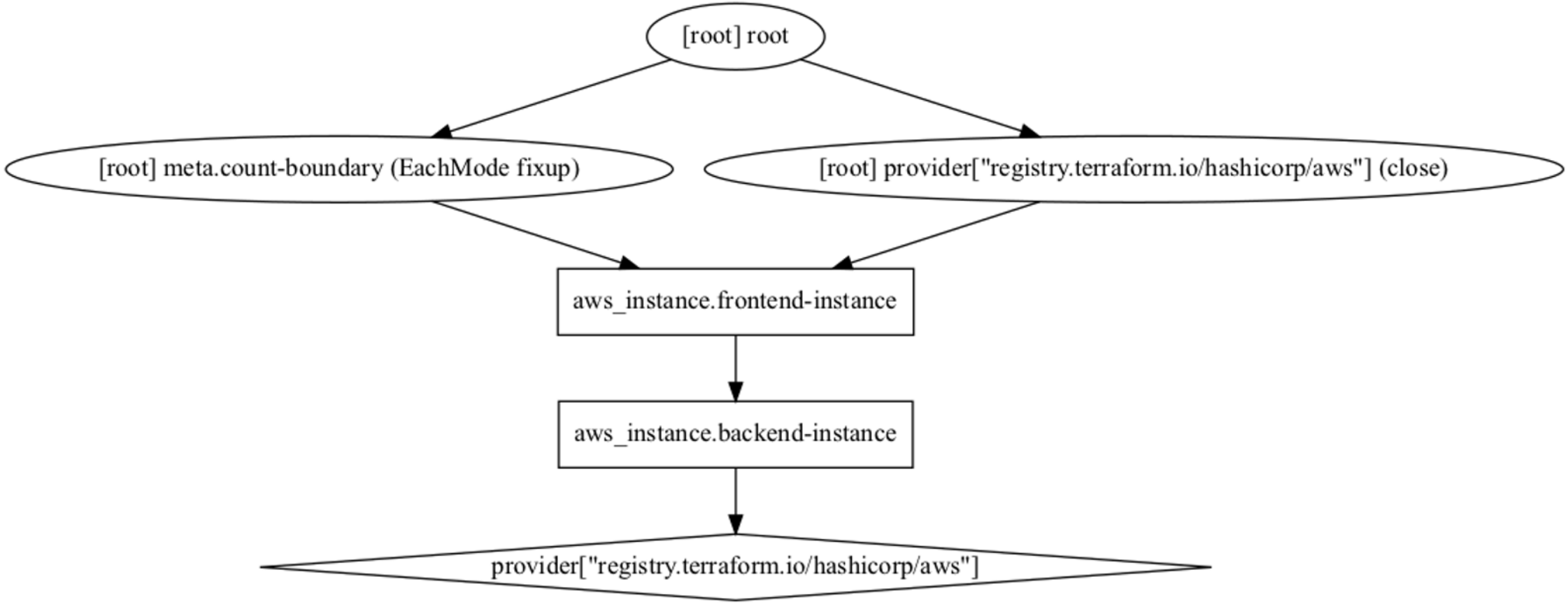
# No dependency graph representation

```
$ terraform graph | dot -Tpng > no-dep-graph.png
```



# Dependency graph representation

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



# S3 Storage in AWS

# S3 Bucket-Ressource

```
resource "aws_s3_bucket" "bucket" {
  bucket = var.s3bucket_name
  acl    = "private"

  versioning {
    enabled = true
  }
}
```

# S3 Bucket Upload

```
resource "aws_s3_bucket_object" "video_upload_object" {
  bucket = var.s3bucket_name
  key    = "video.mp4"
  source = "video.mp4"

  # The filemd5() function is available in Terraform 0.11.12 and later
  # For Terraform 0.11.11 and earlier, use the md5() function and the file() function:
  # etag = "${md5(file("path/to/file"))}"
  etag = filemd5("video.mp4")

  depends_on = [
    aws_s3_bucket.bucket
  ]

}
```

# S3 Hosting a Websiterandom

```
resource "aws_s3_bucket" "static_bucket" {
  bucket = var.static_bucket_name
  acl    = "public-read"

  website {
    index_document = "index.html"
    error_document = "error.html"
  }
}
```

# S3 Create a Policy / using templatefile()

```
resource "aws_s3_bucket_policy" "bucket_policy" {
  bucket = aws_s3_bucket.static_bucket.id

  policy = templatefile("bucket_policy.tpl", {
    bucket_name = var.static_bucket_name
  })
}
```

# Create a Policy

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Sid": "Allow Public Access to All Objects",
      "Effect": "Allow",
      "Principal": "*",
      "Action": "s3:GetObject",
      "Resource": "arn:aws:s3:::${bucket_name}/*"
    }
  ]
}
```

# Upload a website-Assets

```
resource "aws_s3_bucket_object" "html_object" {
  bucket       = var.static_bucket_name
  key          = "index.html"
  source       = "index.html"
  content_type = "text/html"

  # The filemd5() function is available in Terraform 0.11.12 and later
  # For Terraform 0.11.11 and earlier, use the md5() function and the file() function:
  # etag = "${md5(file("path/to/file"))}"
  etag = filemd5("index.html")

  depends_on = [
    aws_s3_bucket_policy.bucket_policy,
    aws_s3_bucket.static_bucket
  ]

}
```

# possible HA-Solution

- EC2-Instance´s
- Webserver Installation
- AutoScaling Group
- Application Load Balancer

# Auto Scaling Group

```
resource "aws_autoscaling_group" "autoscaling_group" {
launch_configuration = "${aws_launch_configuration.launch_config.id}"
  min_size             = "${var.autoscaling_group_min_size}"
max_size = "${var.autoscaling_group_max_size}"
  target_group_arns    = ["${aws_alb_target_group.group.arn}"]
vpc_zone_identifier = ["${aws_subnet.main.*.id}"]

tag {
key = "Name"
value = "terraform-example-autoscaling-group"
propagate_at_launch = true
}
}
```

# AWS Load Balancer

- Classic Load Balancer
- Application Load Balancer
- Network Load Balancer

# Application-Loadbalancer Ressource

# LAB

## a simple cluster

- create 3 webserver replicas
- add ingress-rule for port 80 open to the world
- create the ressource loadbalancer aka. classic-lb

# Remote State File

Stores the state at a key in a bucket on Amazon S3

backend also supports state locking via Dynamo DB

# LAB

## Setup your VPC (Module-Support)

- create a VPC in eu-central-1
- create 3 public Subnets for 3 AZ
- create an Internet-GW
- create a Route Table

# LAB

## create an alb-cluster

- create 3 webserver replicas
- add ingress-rule for port 80 open to the world
- add egress-rule for everything open to the world
- create an alb with target-groups

# CUTOUT

# What are variables?

Variables are buckets that store data. Variable declarations looks like this:

```
<datatype> <variable name>; // Initializing <variable name>
<variable name> = <value>; // Assigning <variable name>

// Is equivalant to:
<datatype> <variable name> = <value>; // Initializing and assigning <variable_name>
```

Here are some examples of variable declarations.

```
int myAge = 23; // <datatype> is int (integer)
double pi = 3.14; // <datatype> is double (decimal)
boolean programmingIsAwesome;
programmingIsAwesome = true; // <datatype> is boolean (true/false)
char bestLetter = 'J'; // <datatype> is char (a letter)
```