Terraform

* Why we use Terraform and not Chef, Puppet, Ansible, SaltStack, or CloudFormation
* An introduction to Terraform
* How to manage Terraform state
* How to create reusable infrastructure with Terraform modules
* Terraform tips & tricks: loops, if-statements, and pitfalls
* How to use Terraform as a team

Why infrastructure-as-code?

A long time ago, in a data center far, far away, an ancient group of powerful beings known as sysadmins used to deploy infrastructure manually. Every server, every route table entry, every database configuration, and every load balancer was created and managed by hand. It was a dark and fearful age: fear of downtime, fear of accidental misconfiguration, fear of slow and fragile deployments, and fear of what would happen if the sysadmins fell to the dark side (i.e. took a vacation). The good news is that thanks to the DevOps Rebel Alliance, we now have a better way to do things: Infrastructure-as-Code (IAC).

Instead of clicking around a web UI or SSHing to a server and manually executing commands, the idea behind IAC is to write code to define, provision, and manage your infrastructure. This has a number of benefits:

You can automate your entire provisioning and deployment process, which makes it much faster and more reliable than any manual process.

You can represent the state of your infrastructure in source files that anyone can read rather than a sysadmin’s head.

You can store those source files in version control, which means the entire history of your infrastructure is now captured in the commit log, which you can use to debug problems, and if necessary, roll back to older versions.

You can validate each infrastructure change through code reviews and automated tests.

You can create reusable, documented, battle-tested infrastructure packages that make it easier to scale and evolve your infrastructure.

There is one other very important, and often overlooked, reason for why you should use IAC: it makes developers happy. Deploying code is a repetitive and tedious task. A computer can do that sort of thing quickly and reliably, but a human will be slow and error prone. Moreover, a developer will resent that type of work, as it involves no creativity, no challenge, and no recognition. You could deploy code perfectly for months, and no one will take notice — until that one day where you mess it up.

That creates a stressful and unpleasant environment. IAC offers a better alternative that allows computers to do what they do best (automation) and developers to do what they do best (coding).

Why Terraform?

IAAC infrastructure as code

Compare to cloud formation

CF is in json and little complex.

CF is slow

Exiting device or new device outside CF cannot be managed with cf.

Detailed plan is not visible

Demo

Main.tf

# ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

# CREATE ALL THE RESOURCES TO DEPLOY AN APP IN AN AUTO SCALING GROUP WITH AN ELB

# This template runs a simple "Hello, World" web server in Auto Scaling Group (ASG) with an Elastic Load Balancer

# (ELB) in front of it to distribute traffic across the EC2 Instances in the ASG.

# ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

# ------------------------------------------------------------------------------

# CONFIGURE OUR AWS CONNECTION

# ------------------------------------------------------------------------------

provider "aws" {

region = "us-east-1"

}

# ---------------------------------------------------------------------------------------------------------------------

# GET THE LIST OF AVAILABILITY ZONES IN THE CURRENT REGION

# Every AWS accout has slightly different availability zones in each region. For example, one account might have

# us-east-1a, us-east-1b, and us-east-1c, while another will have us-east-1a, us-east-1b, and us-east-1d. This resource

# queries AWS to fetch the list for the current account and region.

# ---------------------------------------------------------------------------------------------------------------------

data "aws\_availability\_zones" "all" {}

# ---------------------------------------------------------------------------------------------------------------------

# CREATE THE AUTO SCALING GROUP

# ---------------------------------------------------------------------------------------------------------------------

resource "aws\_autoscaling\_group" "example" {

launch\_configuration = "${aws\_launch\_configuration.example.id}"

availability\_zones = ["${data.aws\_availability\_zones.all.names}"]

min\_size = 2

max\_size = 10

load\_balancers = ["${aws\_elb.example.name}"]

health\_check\_type = "ELB"

tag {

key = "Name"

value = "terraform-asg-example"

propagate\_at\_launch = true

}

}

# ---------------------------------------------------------------------------------------------------------------------

# CREATE A LAUNCH CONFIGURATION THAT DEFINES EACH EC2 INSTANCE IN THE ASG

# ---------------------------------------------------------------------------------------------------------------------

resource "aws\_launch\_configuration" "example" {

# Ubuntu Server 14.04 LTS (HVM), SSD Volume Type in us-east-1

image\_id = "ami-2d39803a"

instance\_type = "t2.micro"

security\_groups = ["${aws\_security\_group.instance.id}"]

user\_data = <<-EOF

#!/bin/bash

echo "Hello, World" > index.html

nohup busybox httpd -f -p "${var.server\_port}" &

EOF

# Important note: whenever using a launch configuration with an auto scaling group, you must set

# create\_before\_destroy = true. However, as soon as you set create\_before\_destroy = true in one resource, you must

# also set it in every resource that it depends on, or you'll get an error about cyclic dependencies (especially when

# removing resources). For more info, see:

#

# <https://www.terraform.io/docs/providers/aws/r/launch_configuration.html>

# <https://terraform.io/docs/configuration/resources.html>

lifecycle {

create\_before\_destroy = true

}

}

# ---------------------------------------------------------------------------------------------------------------------

# CREATE THE SECURITY GROUP THAT'S APPLIED TO EACH EC2 INSTANCE IN THE ASG

# ---------------------------------------------------------------------------------------------------------------------

resource "aws\_security\_group" "instance" {

name = "terraform-example-instance"

# Inbound HTTP from anywhere

ingress {

from\_port = "${var.server\_port}"

to\_port = "${var.server\_port}"

protocol = "tcp"

cidr\_blocks = ["0.0.0.0/0"]

}

# aws\_launch\_configuration.launch\_configuration in this module sets create\_before\_destroy to true, which means

# everything it depends on, including this resource, must set it as well, or you'll get cyclic dependency errors

# when you try to do a terraform destroy.

lifecycle {

create\_before\_destroy = true

}

}

# ---------------------------------------------------------------------------------------------------------------------

# CREATE AN ELB TO ROUTE TRAFFIC ACROSS THE AUTO SCALING GROUP

# ---------------------------------------------------------------------------------------------------------------------

resource "aws\_elb" "example" {

name = "terraform-asg-example"

security\_groups = ["${aws\_security\_group.elb.id}"]

availability\_zones = ["${data.aws\_availability\_zones.all.names}"]

health\_check {

healthy\_threshold = 2

unhealthy\_threshold = 2

timeout = 3

interval = 30

target = "HTTP:${var.server\_port}/"

}

# This adds a listener for incoming HTTP requests.

listener {

lb\_port = 80

lb\_protocol = "http"

instance\_port = "${var.server\_port}"

instance\_protocol = "http"

}

}

# ---------------------------------------------------------------------------------------------------------------------

# CREATE A SECURITY GROUP THAT CONTROLS WHAT TRAFFIC AN GO IN AND OUT OF THE ELB

# ---------------------------------------------------------------------------------------------------------------------

resource "aws\_security\_group" "elb" {

name = "terraform-example-elb"

# Allow all outbound

egress {

from\_port = 0

to\_port = 0

protocol = "-1"

cidr\_blocks = ["0.0.0.0/0"]

}

# Inbound HTTP from anywhere

ingress {

from\_port = 80

to\_port = 80

protocol = "tcp"

cidr\_blocks = ["0.0.0.0/0"]

}

}

Terraform

init

plan

Apply

Destroy

Ref

https://www.youtube.com/watch?v=16XHBFy5shw&list=PL\_OdF9Z6GmVaRD6e6sYLQO\_WYqTKcj3aj&index=9