Terraform – Introduction

What is Terraform?

- Terraform is a tool to manage and describe infrastructure
- Terraform is *AAS agnostic
  - It supports many providers for various types of infrastructure
  - You can add your own extensions
  - It does not provide an abstraction layer on top

Why use terraform?

- Terraform is intended to support anything which offers an API
- It supports
  - Virtual machines
    - Openstack, Digital Rebar, The Foreman, CoreOS, …
  - Networking
    - SDN, NFV
    - Cisco ACI, UCS
    - F5 Load balancers
    - …
  - Monitoring (Icinga2)
  - …
  - See https://www.terraform.io/docs/providers/index.html for details
- Terraform catches infrastructure configuration drift

Syntax

- Terraform comes with it’s own DSL
  - Custom language (Hashicorp Configuration Language/HCL)
- HCL is declarative
  - You describe what you want
  - This has limitations(*)
- Supports basic data types
  - Booleans
  - Strings
  - Arrays
  - Maps/Hashes/Key-Value pairs

Filesystem layout

- Terraform will load all files with names ending in ‘.tf’
- A file named ‘main.tf’ is mandatory
- A module named foo:
  - ./foo/main.tf

    /variables.tf
    /outputs.tf

Resources

Terraform works primarily with resources

A resource describes a single logical component of infrastructure and is identified by the “type” + “name” pair

```
resource “type” “name” {
        key = value
        key {
                key = value
                key { … }
        }
}
```

Parameters

- Resources can accept parameters, which allows for some deduplication of code.
  - Staging vs production instances for example

```
variable “disk” {
	default = 500
	description = “Default disk size in GB”
}

resource "google_compute_instance_template" "instance_template" {
	disk {
		disk_size_gb = “${var.disk}”
	}
}
```

Modules

A module is a collection of resources

```
module “foo” {
     source = ./resource
     param = ...
}
```

```
module “staging_host” {
     source = ./host
     disk = 10
}
module “testing_host” {
     source = ./host
     disk = 200
}
module “production_host” {
     source = ./host
}
```

Outputs

- Terraform modules can provide outputs
  - These can be referenced by other resources
  - They are useful in building dependency trees

State

- Terraform maintains global state for a system
  - This includes all resources managed by Terraform
  - This is effectively a CMDB with dependencies listed
- Terraform state defaults to being local
  - For people in teams, shared state is recommended
    - Unless you can guarantee only one user at a time
- It is possible to store state remotely

Backends

- Remote state is stored using a “backend”
- Most backends support state locking
- This is extremely useful when storing common state between modules (outputs, networks and the like)

```
terraform {
    backend "gcs" {
        prefix = "my-awesome-project"
    }
}

provider "google" {
    credentials = "${file("../../account.json")}"
    region      = "europe-west1"
    project     = "my-awesome-project"
}
```

Data sources and local variables

- Data sources allow data to be fetched or computed for use elsewhere in Terraform configuration.
- Locals act as local variables

```
data "terraform_remote_state" "department" {
    backend = "gcs"
    config {
        prefix = "department"
        credentials = "${var.credentials}"
        bucket = "${var.bucket}"
        encryption_key = "${var.encryption_key}"
    }
}

locals {
    subnet = "${data.terraform_remote_state.department.my_subnet}"
    subnet_project = "department"
}
```

Using terraform

- Terraform is provided as a single, cross-platform go binary
- Terraform workflow is roughly:
  terraform init # initial setup of state and backends
  terraform plan # make plan from state and local changes
  terraform apply # Actual change
- Repeat the plan and apply steps each time you need to make a configuration change