DevOps Fundamentals

Infra As Code

Agenda

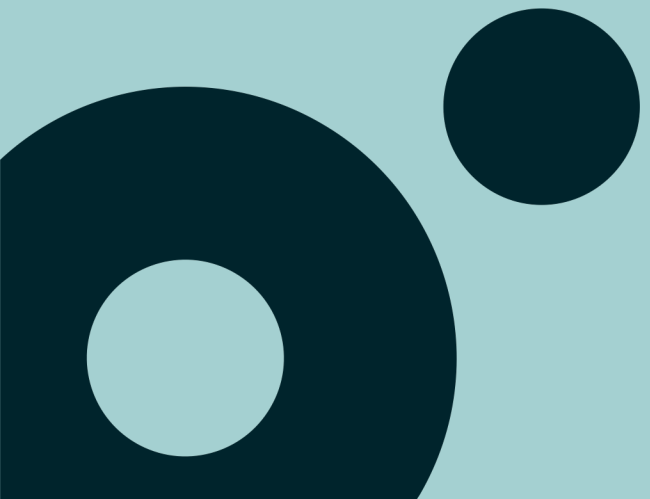
Infra as Code Principles

Terraform

Infra as Code on CI/CD

moOngy.

Infra As Code Principles



What is Infra As Code (IaC)?

Infrastructure as code is the approach to defining your infrastructure through source code that can then be treated just like any software system

Infrastructure can be computing (like VMs), networking, security and any cloud managed service and resource (like Kubernetes clusters, serverless, etc.)

This code (as any type of code) must be kept in source control to allow auditability, versioning all full integration with CI/CD

Natural practice with cloud computing but can be use on several on-prem virtual environments

IaC: Benefits

Faster and easier way to provisioning, validate and reconfigure your infra

Help on configuration drift (consistency)

Control cost on dynamic environments

Full integration with source control

Versioned together with source code (and pipelines ☺)

Serves as infrastructure live documentation using declarative configuration

Easy and recommended integration with CI/CD process, adding additional layer of security

Allow you to test your infra definition

IaC: Declarative configuration

Declarative configuration allow to define desired state on a more human-readable style

You define **what** you want to achieve at the end

How to implement your configuration is not your concern. Let the tooling do that for you

Opposite of imperative configuration like scripting where you need to define all the steps

Your configuration is **idempotent**, means you may ask to get your desired state as much you need and at the end you get always the same outcome

With imperative configuration you may get the same but you need to do it by yourself

IaC vs Configuration Management

Infrastructure as code defines all your infra resources (computing, networking, managed services)

Configuration management defines configuration inside your infra (software to be installed on a specific machine)

Both may (or must…) be used together to allow you to a complete dynamic and automated way to manage all your infrastructure and its configuration

Most used tooling for Configuration Management are Ansible, Chef and Puppet

Most of Configuration Management tools allow you to create infrastructure too but with limited capabilities

IaC: Tooling

Two main flavors: proprietary or provider-agnostic

Proprietary is related and managed only by a specific provider

- Azure have ARM (Azure Resource Manager) Templates and Bicep
- AWS has Cloud Formation
- GCP has Google Deployment Manager

Provider-agnostic tooling is not tightly coupled with cloud (or on-prem) provider you want to use

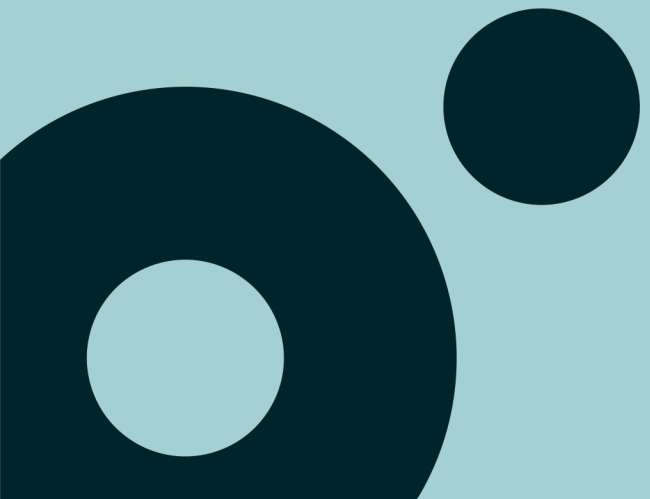
- Terraform
- Pulumi
- Ansible

IaC: Tooling

	Pros	Cons
Proprietary	Always updated with last features Direct support from provider	Limited to one Provider You may need to learn several tools
Provider-agnostic	Better on hybrid environments Bigger Communities	Feature parity Changing Provider is not only a configuration task

moOngy.

Terraform



What is Terraform

Multi platform and multi provider IaC tooling from Hashicorp

Biggest community with a big ecosystem of providers

Provides a clean and easy way to write and maintain your code

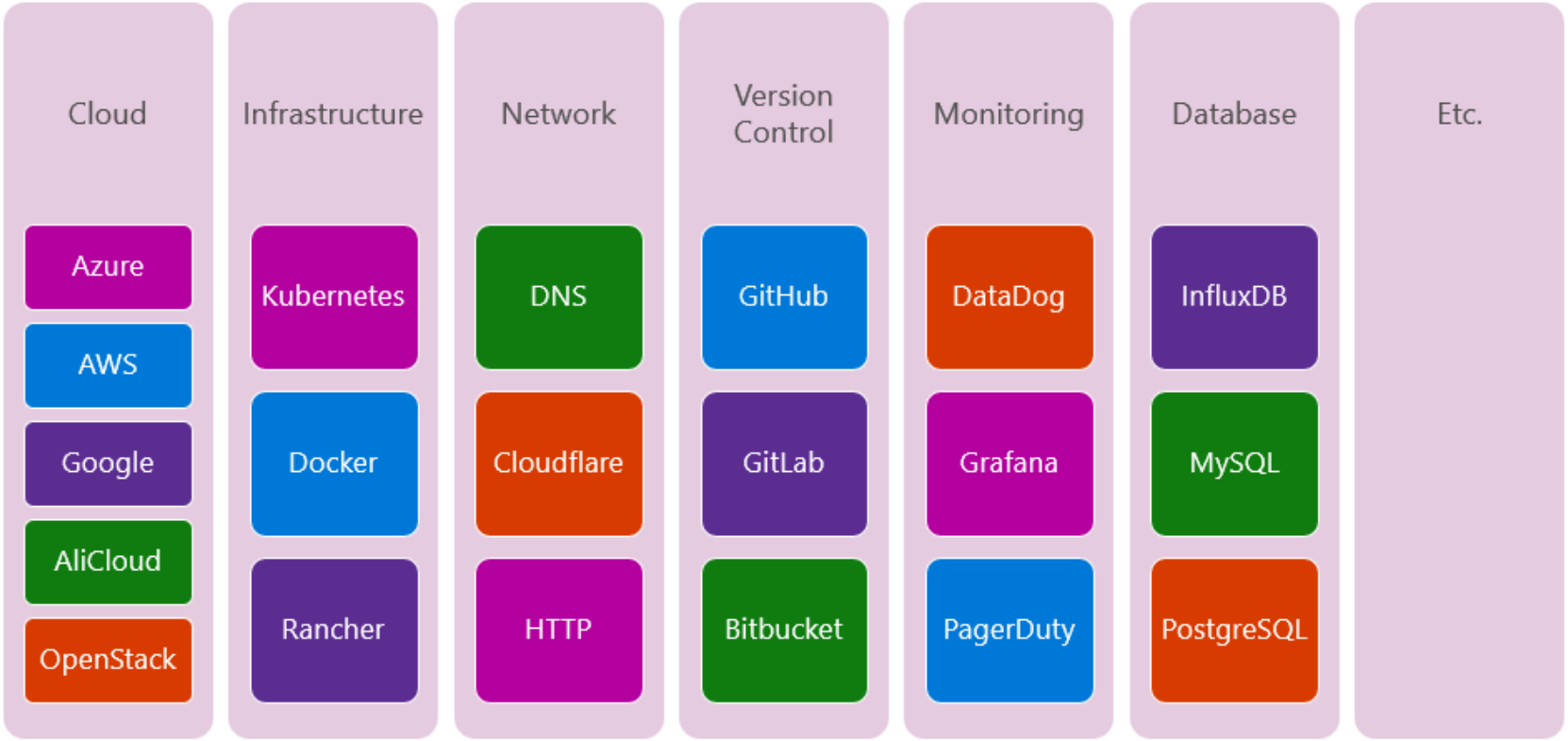
Uses a proprietary language (HCL) but similar with JSON/YAML

moOngy.

Terraform Providers

Big ecosystem of providers (Browse Providers | Terraform Registry)

Allow to everyone defines your own provider if it not exists



moOngy.

Terraform: How to start?

As other DevOps tooling, you may use a CLI

It ships as a single binary which is written in Go. Terraform is cross platform and can run on Linux, Windows, or MacOS.

Installing terraform is easy. You simply download a zip file, unzip it, and run it.

Terraform: HashiCorp Configuration Language (HCL)

The HashiCorp Configuration Language (HCL) is a small domain specific language which is based on JSON.

```
resource "azurerm_redis_cache" "sample" {
  name                = "tf-redis-basic"
  location            = "${azurerm_resource_group.test.location}"
  resource_group_name = "${azurerm_resource_group.test.name}"
  capacity            = 0
  family              = "C"
  sku_name            = "Basic"
  enable_non_ssl_port = "${var.redis_enable_non_ssl}"
  tags                = "${local.all_tags}"
}
```

moOngy.

Terraform: Terraform CLI

```
# Basic Terraform Commands
terraform version
terraform help
terraform init
terraform plan
terraform apply
terraform destroy
```

Terraform: Basic commands

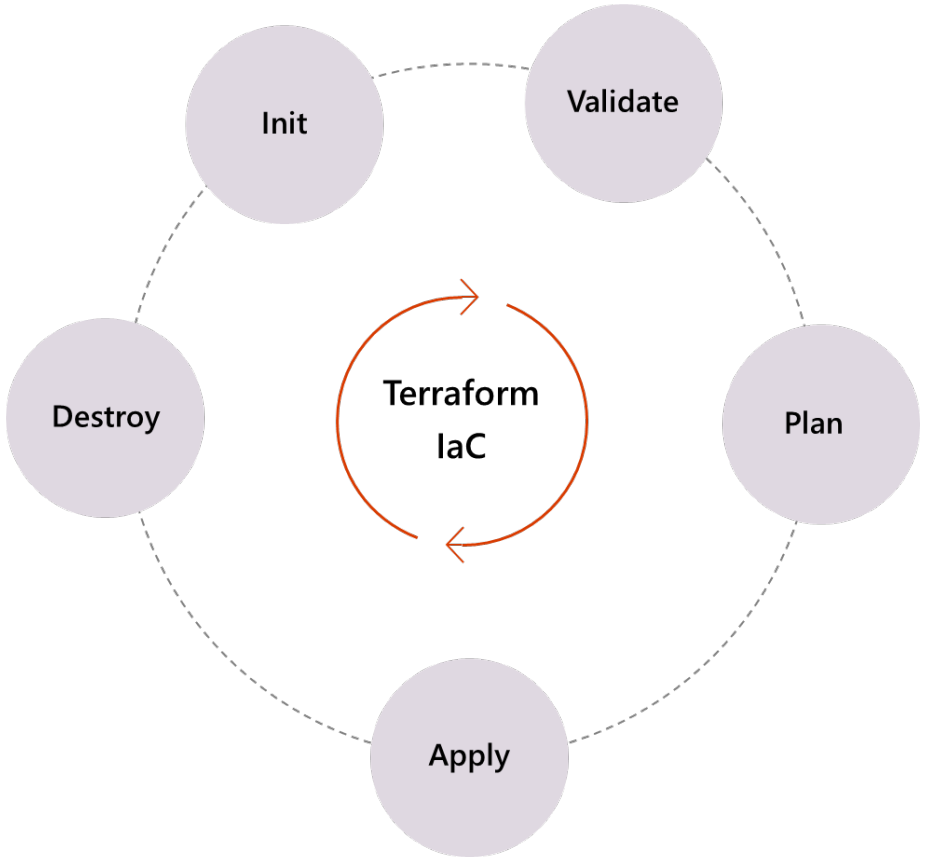
Init: Initialize a working directory with Terraform configuration files

Validate: Validates configuration files in a directory without checking remotely

Plan: It creates an execution plan (aka WhatIf)

Apply: Deploy the changes required to reach the desired state

Destroy: Remove the TF manage infrastructure



moOngy.

Terraform Init

Terraform fetches any required providers and modules and stores them in the .terraform directory.

If you add, change or update your modules or providers you will need to run init again.

```
$ terraform init
Initializing the backend...
Initializing provider plugins...
- Checking for available provider plugins...
- Downloading plugin for provider "azurerm" (hashicorp/azurerm) 1.35.0.
...
provider.azurerm: version = "~> 1.35"

Terraform has been successfully initialized!
```

moOngy.

Terraform Validate

`terraform validate` checks if your terraform files are valid without making any remote validation

```
> terraform validate

Warning: "skip_credentials_validation": [DEPRECATED] This field is deprecated and will
 be removed in version 3.0 of the Azure Provider


Success! The configuration is valid, but there were some validation warnings as shown
above.
```

Terraform Plan

Preview your changes with **`terraform plan`** before you apply them.

```
$ terraform plan
An execution plan has been generated and is shown below.
Terraform will perform the following actions:
  # azurerm_resource_group.myresourcegroup will be created
  + resource "azurerm_resource_group" "myresourcegroup" {
      + id       = (known after apply)
      + location = "centralus"
      + name     = "bugsbunny-workshop"
      + tags     = (known after apply)
    }
Plan: 1 to add, 0 to change, 0 to destroy.
```

moOngy.

Terraform Apply

`terraform apply` runs a plan and then if you approve, it applies the changes

```
$ terraform apply
An execution plan has been generated and is shown below.

Terraform will perform the following actions:
  # azurerm_resource_group.myresourcegroup will be created
  + resource "azurerm_resource_group" "myresourcegroup" {
      + id       = (known after apply)
      + location = "centralus"
      + name     = "seanc-workshop"
      + tags     = (known after apply)
    }

Plan: 1 to add, 0 to change, 0 to destroy.
```

Terraform Destroy

`terraform destroy` does the opposite. If you approve, your infrastructure is destroyed.

```
$ terraform destroy
An execution plan has been generated and is shown below.

Terraform will perform the following actions:

  # azurerm_resource_group.myresourcegroup will be destroyed
  - resource "azurerm_resource_group" "myresourcegroup" {
      - id       = "/subscriptions/14692f20-9428-451b-8298-102ed4e39c2a/resourceGroups/seanc-
      - location = "centralus" -> null
      - name     = "seanc-workshop" -> null
      - tags     = {} -> null
    }

Plan: 0 to add, 0 to change, 1 to destroy.
```

moOngy.

Terraform: Your first script

Terraform uses the concept of resources

`resource` = Top level keyword

`type` = Type of resource. Ex: azurerm_virtual_machine.

`name` = Arbitrary name to refer to this resource. Used internally by terraform. This field cannot be a variable.

```
resource "type" "name" {
  parameter = "foo"
  parameter2 = "bar"
  list = ["one", "two", "three"]
}
```

moOngy.

Demo: First Script Terraform

Terraform: Variables

Variables allow to bring more dynamic behavior to your Terraform code

Defined on a ‘tf’ file work as variables and parameters

Variables can have default settings

If you omit the default, the user will be prompted to enter a value

Terraform: Variables

Variables must have a “name” property

Variables can have a “description” and “default” properties

```
variable "prefix" {
  description = "This prefix will be included in the name of most resou
}

variable "location" {
  description = "The region where the virtual network is created."
  default     = "centralus"
}
```

Terraform: How to set Variables?

1. Command line flag - run as a command line switch
2. Configuration file - set in your terraform.tfvars file
3. Environment variable - part of your shell environment
4. Default Config - default value in variables.tf
5. User manual entry - if not specified, prompt the user for entry

Goes from highest precedence (1) to lowest (5)

Terraform: Outputs

This is where you configure any messages or data you want to show at the end of a terraform apply

```
output "Vault_Server_URL" {
  value = "http://${azurerm_public_ip.vault-pip.fqdn}:8200"
}

output "MySQL_Server_FQDN" {
  value = azurerm_mysql_server.mysql.fqdn
}

output "catapp_url" {
  value = "http://${azurerm_public_ip.catapp-pip.fqdn}"
}
```

Terraform: How to organize my code?

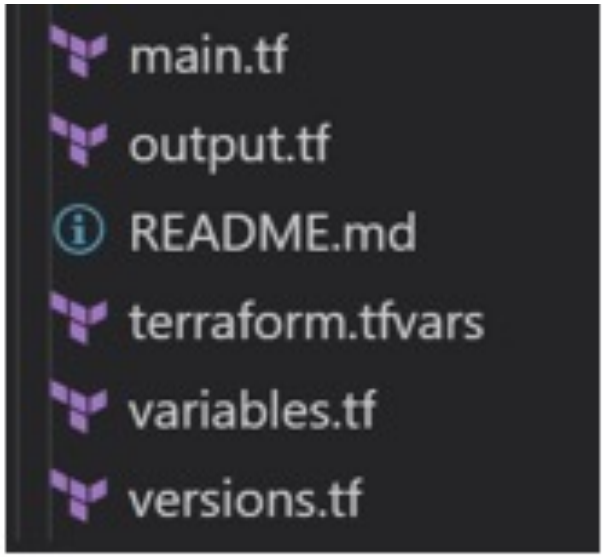
main.tf (Required): Main structure in order to deploy the complete infrastructure directly or through calls to modules

output.tf: Identify each return value of a Terraform module

terraform.tfvars: Assign values to variables in a file instead of arguments in a command

variables.tf: Define variables required (or not) in main.tf, we need to define the var type and it allows to you assign a default value.

versions.tf: Define minimum version required for terraform and providers in a module



moOngy.

Terraform: Dependency Mapping

Terraform can automatically keep track of dependencies for you

```
resource "azurerm_resource_group" "hashitraining" {
  name     = "${var.prefix}-vault-workshop"
  location = var.location
}

resource "azurerm_virtual_network" "vnet" {
  name                = "${var.prefix}-vnet"
  location            = azurerm_resource_group.hashitraining.location
  address_space       = [var.address_space]
  resource_group_name = azurerm_resource_group.hashitraining.name
}
```

moOngy.

Terraform State

Terraform is a stateful application

This means that it keeps track of everything you build inside of a state file.

The terraform.tfstate and terraform.tfstate.backup files that appeared inside your working directory.

The state file is Terraform's source of record for everything it knows about.

Terraform State

Whenever you run a plan or apply, Terraform reconciles three different data sources

- What you wrote in your code
- The state file
- What actually exists

Here are the four different things that can happen to each resource during a plan/apply

```
+   create
-   destroy
-/+ replace
~   update in-place
```

moOngy.

Terraform Backends

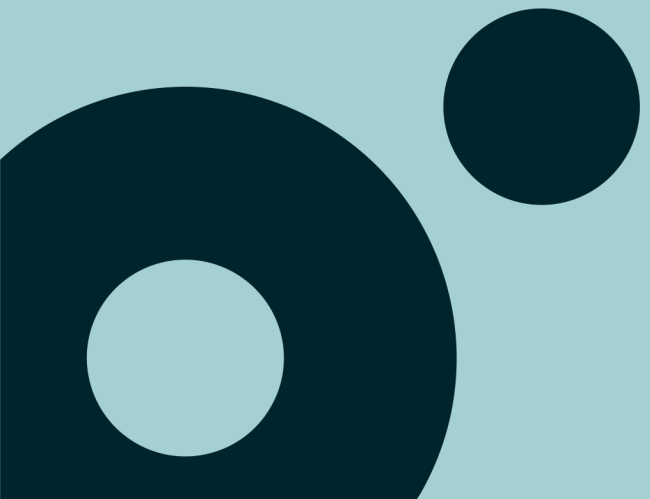
Connect terraform with external storage to keep your tfstate file

Enhanced Backends: Uses Terraform Cloud

Standard Backends: Uses other providers (Azure, AWS, ...)

moOngy.

Infra As Code on CI/CD



Automate IaC

Infra as Code is a practice that you must always (as possible) on your CI/CD practices

Several benefits

- Allow you to validate your infra anytime you want to make a deploy
- Automate infra provisioning/configuration
- Create/destroy specific environments only needed for a short period of time

moOngy.

Demo: IaC on GitHub Actions

Lab

Lab 05 - IaC

Learning Objectives

- Create a new Azure Storage Account to keep your Terraform state
- Create Terraform Scripts to deploy your infrastructure
- Update your workflows to include Terraform execution

Markdown version: https://github.com/tasb/devops-with-github-training/blob/main/labs/lab05.md

HTML version: https://tasb.github.io/devops-with-github-training/labs/lab05.html

moOngy.
Minds on the move

Rua Sousa Martins, nº 10

1050-218 Lisboa | Portugal