Best Practices with Terraform Scripts

Robert Rozas Navarro
Premier Field Engineer
Apps Domain

Agenda

1. Best Practices
2. Hands on Lab 2
3. Q&A

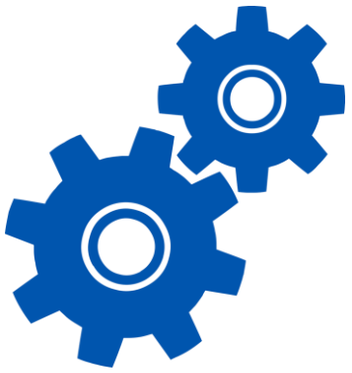
Terraform
Best Practices

Best Practice 1

- Use remote backend
- Your laptop is no place for your infrastructure source of truth
- Use **data sources** and **terraform_remote_state** specifically as a glue between infrastructure modules within composition
- Managing a **tfstate** file in git is a nightmare
- Later when infrastructure layers starts to grow in any direction (number of dependencies or resources)

Best Practice 1 (..Continued..)

- Using the backend functionality has definitely benefits:
- Working in a team: it allows for collaboration, the remote state will always be available for the whole team
- The state file is note stored locally. Possible sensitive information is now only stored in the remote state
- Some backends will enable remote operations. The terraform apply will then run completely remote. These are called the enhanced backends



Best Practice 1 (..Continued..)

- You can also store your state in Azure Storage:

```
terraform {
    backend "azurerm" {
        resource_group_name  = "StorageAccount-ResourceGroup"
        storage_account_name = "abcd1234"
        container_name       = "tfstate"
        key                  = "prod.terraform.tfstate"
    }
}
```

Best Practice 1 (..Continued..)

- When using an Azure remote state, it's best to configure the Service Principal credentials:

```
$ az ad sp create-for-rbac --role="Contributor" --scopes="/subscriptions/SUBSCRIPTION_ID"
```

This command will output 5 values:

```
{
  "appId": "00000000-0000-0000-0000-000000000000",
  "displayName": "azure-cli-2017-06-05-10-41-15",
  "name": "http://azure-cli-2017-06-05-10-41-15",
  "password": "0000-0000-0000-0000-000000000000",
  "tenant": "00000000-0000-0000-0000-000000000000"
}
```

Best Practice 2

- Manage Terraform, Azure provider and modules version
- While individual resources are like atoms in the infrastructure, resource modules are molecules. Module is a smallest versioned and shareable unit.
- Examples and Terraform modules should contain documentation explaining features and how to use them.
- Infrastructure modules and compositions should persist their state in a remote location which can be reached by others in a controllable way

Best Practice 2 (..Continued..)

- You can use modules to make your terraform project more organized
- Infrastructure modules and compositions should persist their state in a remote location which can be reached by others in a controllable way
- Use third party modules (i.e from github)
- Allows you to reuse parts of your code

Best Practice 2 (..Continued..)

Use a module from git

```
module "module-example" {
    source = "github.com/wardviaene/terraform-module-example"
}
```

Use a module from a local folder

```
module "module-example" {
    source = "./module-example"
}
```

Best Practice 2 (..Continued..)

Pass arguments to the module

```
module "module-example" {
    source = "./module-example"
    region = "us-west-1"
    ip-range = "10.0.0.0/8"
    cluster-size = "3"
}
```

Best Practice 2 (..Continued..)

Inside the module folder, you just have again, terraform files:

module-example/vars.tf

```
variable "vm_os_simple" {
  default = ""
}

# Definition of the standard OS with "SimpleName" = "publisher,offer,sku"
variable "standard_os" {
  default = {
    "UbuntuServer"   = "Canonical,UbuntuServer,16.04-LTS"
    "WindowsServer" = "MicrosoftWindowsServer,WindowsServer,2016-Datacenter"
    "RHEL"           = "RedHat,RHEL,7.5"
    "openSUSE-Leap" = "SUSE,openSUSE-Leap,42.2"
    "CentOS"         = "OpenLogic,CentOS,7.6"
    "Debian"         = "credativ,Debian,8"
    "CoreOS"         = "CoreOS,CoreOS,Stable"
    "SLES"           = "SUSE,SLES,12-SP2"
  }
}
```

module-example/cluster.tf

```
resource "azurerm_virtual_machine" "vm-linux" {
  count                         = ! contains(list(var.vm_os_simple, var.vm_os_offer), "Windows")
  name                          = "${var.vm_hostname}${count.index}"
  location                      = var.location
  resource_group_name           = azurerm_resource_group.vm.name
  availability_set_id           = azurerm_availability_set.vm.id
  vm_size                       = var.vm_size
  network_interface_ids         = [element(azurerm_network_interface.vm.*.id, count.index)]
  delete_os_disk_on_termination = var.delete_os_disk_on_termination
```

Best Practice 3

- Use implicit dependencies
- Implicit dependencies should be used whenever possible (see this article from terraform.io website for more information).
- With IaC the resources will be configured exactly as declared, and implicit dependencies can be used to ensure the creation order.

Best Practice 4

- Try to practice a consistent structure and **naming convention**
- Use _ (underscore) instead of - (dash) in all: resource names, data source names, variable names, outputs.
- Only use lowercase letters and numbers.

Generating Images

- The output of terraform graph is in the DOT format, which can easily be converted to an image by making use of dot provided by GraphViz:

```
$ terraform graph | dot -Tsvg > graph.svg
```

How should I structure my Terraform configurations?

- What is the complexity of your project?
- How often does your infrastructure change?
- How environments are grouped?
- Do not store all your code in a single file
- Provide a file structure that allows “separation of concerns”
- You can try more than one way to do it.

Terraform configurations (..Continued..)

provider.tf

```
provider "azurerm" {
  # Whilst version is optional, we /strongly recommend/ using it
  version = "=1.38.0"

  subscription_id = "00000000-0000-0000-0000-000000000000"
  client_id       = "00000000-0000-0000-0000-000000000000"
  client_secret   = "${var.client_secret}"
  tenant_id       = "00000000-0000-0000-0000-000000000000"
}
```

terraform.tfvars

```
$ export ARM_CLIENT_ID="00000000-0000-0000-0000-000000000000"
$ export ARM_CLIENT_SECRET="00000000-0000-0000-0000-000000000000"
$ export ARM_SUBSCRIPTION_ID="00000000-0000-0000-0000-000000000000"
$ export ARM_TENANT_ID="00000000-0000-0000-0000-000000000000"
```

vars.tf

```
variable "resource_group_name" {
  description = "The name of the resource group in which the resources
  default     = "terraform-compute"
}

variable "location" {
  description = "The location/region where the virtual network is create
}

variable "vnet_subnet_id" {
  description = "The subnet id of the virtual network where the virtual
}
```

instance.tf

```
resource "azurerm_virtual_machine" "vm-linux" {
  count                         = ! contains(list(var.vm_os_simple, var.vm_os_offer), "Windows")
  name                          = "${var.vm_hostname}${count.index}"
  location                      = var.location
  resource_group_name           = azurerm_resource_group.vm.name
  availability_set_id           = azurerm_availability_set.vm.id
  vm_size                       = var.vm_size
  network_interface_ids         = [element(azurerm_network_interface.vm.*.id, count.index)]
  delete_os_disk_on_termination = var.delete_os_disk_on_termination
```

Terraform configurations (..Continued..)

- Putting all code in main.tf is a good idea when you are getting started or writing an example code
- In all other cases you will be better having several files split logically like this:

  main.tf - call modules, locals and data-sources to create all resources

  variables.tf - contains declarations of variables used in main.tf

  outputs.tf - contains outputs from the resources created in main.tf
- terraform.tfvars should not be used anywhere except composition.

Hands On Lab 1

Q&A?