Terraform

(vs. CloudFormation vs. ARM)

VISMA

Terraform

VISMA

What is Terraform / IasC* ?

Terraform is a tool for building, changing, and versioning infrastructure safely and efficiently. Terraform can manage existing and popular service providers as well as custom in-house solutions. (AWS, Azure, GCP, GitHub ...)

more on https://registry.terraform.io/browse/providers

*IasC = Infrastructure as a Code

VISMA

Terraform does not only manage infrastructure but can manage your whole platform including applications !

Terraform is

- open source
- declarative
- stateful
- supports over 100 providers

VISMA

Architecture

You define you configuration via configuration files & variables. You plug in your state file (blank or preexisting). Terraform generates **execution plan** and after confirmation it executes plan on configured providers.

Configuration files

HCL structure (Hashicorp Configuration Language (HCL) is a unique configuration language). All files have ***.tf** extension.

- Providers definition
- Resources definitions / Data Sources
- Variables definitions
- Modules

VISMA

Variables

Input variables serve as parameters for a Terraform module, allowing aspects of the module to be customized without altering the module's own source code, and allowing modules to be shared between different configurations.

Configured via block definition called **variable**

- Type (String, Number, Bool, List, Set, Map, Object, Tuple)
- Default
- Description
- Validation
- Sensitive

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

  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-\"."
  }
}
```

VISMA

Variables

Variable configuration files doesn't need to include default values. If we are having multiple environments / configuration sets we can use **variable definitions files** (**.tfvars**)

prod.tfvars
image_id=ami-0ff8a91507f77f867

dev.tfvars
image_id=ami-0023040df18933030

```
$ terraform apply -var-file="prod.tfvars"
```

VISMA

Variables

Variables on the Command Line

```
$ terraform apply -var="image_id=ami-abc123"
```

Environment Variables

```
$ export TF_VAR_image_id=ami-abc123
```

VISMA

Variables

Variable Definition Precedence

Terraform loads variables in the following order, with later sources taking precedence over earlier ones.

- Environment variables
- The *terraform.tfvars* file, if present.
- The *terraform.tfvars.json* file, if present.
- Any **.auto.tfvars* or **.auto.tfvars.json* files, processed in lexical order of their filenames.
- Any *-var* and *-var-file* options on the command line, in the order they are provided.

VISMA

Variables

Locals declaration

```
locals {
  instance_ids = concat(aws_instance.blue.*.id,
aws_instance.green.*.id)
}
```

```
locals {
  common_tags = {
    Service  = local.service_name
    Owner    = local.owner
  }
}
```

Using Local values

```
resource "aws_instance" "example" {
  # ...

  tags = local.common_tags
}
```

VISMA

Resources

Resources are the most important element in the Terraform language. Each resource block describes one or more infrastructure objects, such as virtual networks, compute instances, or higher-level components such as DNS records.

```
resource "aws_instance" "web" {
  ami           = "ami-a1b2c3d4"
  instance_type = "t2.micro"
}
```

https://registry.terraform.io/browse/providers

VISMA

Resources

The Terraform language defines several meta-arguments, which can be used with any resource type to change the behavior of resources.

The following meta-arguments are documented on separate pages:

- **depends_on**, for specifying hidden dependencies
- **count**, for creating multiple resource instances according to a count
- **for_each**, to create multiple instances according to a map, or set of strings
- **provider**, for selecting a non-default provider configuration
- **lifecycle**, for lifecycle customizations
- **provisioner** and **connection**, for taking extra actions after resource creation

Data sources

Data sources allow data to be fetched or computed for use elsewhere in Terraform configuration. Use of data sources allows a Terraform configuration to make use of information defined outside of Terraform, or defined by another separate Terraform configuration.

```
data "aws_ami" "example" {
  most_recent = true

  owners = ["self"]
  tags = {
    Name   = "app-server"
    Tested = "true"
  }
}
```

VISMA

Providers

Tiers

- Official
- Verified
- Community
- Archived

Providers are written in Go, using the Terraform Plugin SDK.

VISMA

Providers

```
provider "azurerm" {
  version         = "~>2.2.0"
  tenant_id       = "xxxx"
  subscription_id = "xxx"

  features {
    key_vault {
      purge_soft_delete_on_destroy = true
    }
  }
}
```

```
provider "aws" {
  region = var.region
}

provider "aws" {
  alias  = "virginia"
  region = "us-east-1"
}
```

VISMA

Backend configuration

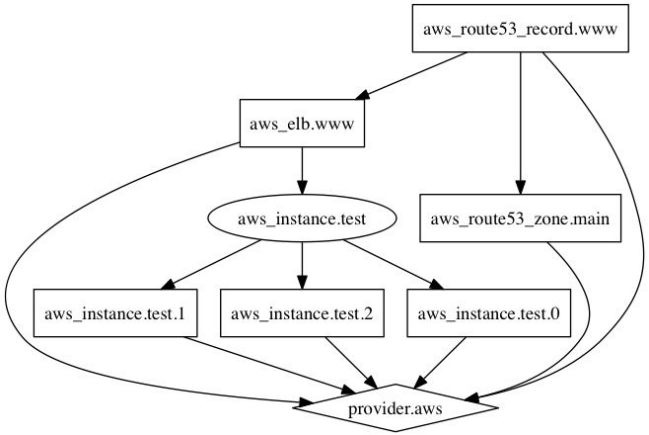
```
terraform {
  required_version = ">= 0.14.0"
  backend "s3" {
    bucket = "sample-project"
    key    = "terraform-dev.tfstate"
    region = "eu-central-1"
  }
}
```

```
terraform {
  backend "azurerm" {
    subscription_id      = "xxx"
    resource_group_name  = "xxx"
    storage_account_name = "xxx"
    container_name       = "tfstate"
    key                  = "terraform-int.tfstate"
  }
}
```

VISMA

Important commands

- init
- plan
- apply
- destroy
- taint
- graph (https://github.com/28mm/blast-radius)



apply and destroy commands support targeting via command line parameter **-target**

VISMA

ARM

What is ARM & ARM Template ?

Azure is managed using an API: Originally it was managed using the Azure Service Management API or ASM which control deployments of what is termed “Classic”. This was replaced by the Azure Resource Manager or ARM API. The resources that the ARM API manages are objects in Azure such as network cards, virtual machines, hosted databases.

The main benefits of the ARM API are that you can deploy several resources together in a single unit and that the deployments are idempotent, in that the user declares the type of resource, what name to use and which properties it should have; the ARM API will then either create a new object that matches those details or change an existing object which has the same name and type to have the same properties.

ARM Templates are a way to declare the objects you want, the types, names and properties in a JSON file which can be checked into source control and managed like any other code file. ARM Templates are what really gives us the ability to roll out Azure “Infrastructure as code”.

Advantages vs. Disadvantages

Advantages

- Strong Microsoft support
- Ability to generate template directly from Azure Portal without writing a single code of line
- Support almost all functionality out of box / does not rely on 3rd party implementations
- They can be digested by Terraform :)

Disadvantages

- Messy JSON
- Only Azure … sorry
- Hard to do Nested templates - not so comprehensive, ease of read, needs declaration
- “Unavailability” of state file for further inspection
- Can’t destroy only part of deployment
- Bad dependency management (DependsON)

VISMA

CloudFormation

VISMA

What is CloudFormation ?

AWS CloudFormation gives you an easy way to model a collection of related AWS and third-party resources, provision them quickly and consistently, and manage them throughout their lifecycles, by treating infrastructure as code. A CloudFormation template describes your desired resources and their dependencies so you can launch and configure them together as a stack. You can use a template to create, update, and delete an entire stack as a single unit, as often as you need to, instead of managing resources individually. You can manage and provision stacks across multiple AWS accounts and AWS Regions.

Advantages vs. Disadvantages

Advantages

- Strong AWS support
- Ability to generate template directly from AWS Console without writing a single code of line
- Support almost all functionality out of box / does not rely on 3rd party implementations
- Support own resource providers implementations
- Automatic rollback of stack

Disadvantages

- Messy YAML / JSON
- Hard to do Nested stacks - not so comprehensive, ease of read, needs declaration
- “Unavailability” of state file for further inspection
- Can’t destroy only part of deployment
- Bad dependency management (DependsON)

VISMA

Lubos Olejar

DevOps Engineer

lubos.olejar@visma.com

VISMA