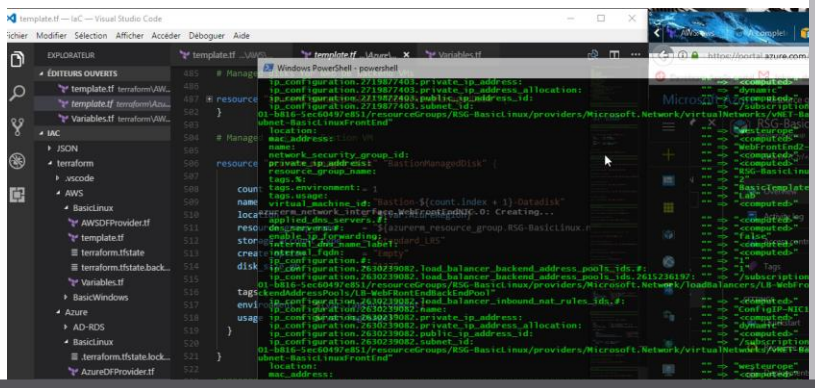


Teknews.cloud

Getting Started with Azure Deployment through Terraform

David Frappart

29/11/2017

Contributor

Name	Title	Email
David Frappart	Cloud Architect	david@dfitcons.info

Version Control

Version	Date	State
1.0	2017/11/27	Final

Table of Contents

1 Introduction to Infrastructure as Code and Terraform

1.1 Infrastructure as Code

1.1.1 Defining Infrastructure as Code

Wikipedia define Infrastructure as code as below:

Infrastructure as code (IaC) is the process of managing and provisioning computer data centers through machine-readable definition files, rather than physical hardware configuration or interactive configuration tools. The IT infrastructure meant by this comprises both physical equipment such as bare-metal servers as well as virtual machines and associated configuration resources. The definitions may be in a version control system. It can use either scripts or declarative definitions, rather than manual processes, but the term is more often used to promote declarative approaches.

As stated in the definition, the objective is to gain agility in Infrastructure deployment by coding the infrastructure. Thus ensuring consistency in environments deployment. Also, by transforming the Infrastructure in chunks of code, it should be easier to have a modular architecture of the infrastructure, allowing IT to move forwrd Lean approach.

1.1.2 The challenge of Infrastructure as Code

In traditional Datacenter, either relying on old-fashioned Physical Servers or heavily virtualized (on the compute side), the IT is generally organized in Towers, each tower responsible of one scope such as Servers, OS, Middleware, Network…
The applications tend to be in a monolithic design, requiring huge effort for each evolution of the application stack.
Accumulation of proprietary solution, often difficult to interoperate, had more complexity to the IT Operations teams which transitively, become unable to provide environment in an appropriate time frame, when a request / project emerges.

In this picture, the idea of coding the infrastructure is an alien concept. First of all, the range of tools and solutions is so broad that it is quite complex to have a complete view of all those tools, extension, and all the API available if any exist.
Secondly, the term “code” is not usually widely popular to IT Operations teams. At best, automation of task is limited to small perimeters, by the use of scripts and classic scheduling ways. At worst, the vast majority of Administration tasks are realized with GUI tools. While GUI give a way to unexperienced IT guys a way to be operational quite rapidly, the gains for a more experienced one are few. By no mean it is possible to reduce indefinitely the time between clicks and to protect against human error, which is bound to happen sometimes.

Infrastructure as Code is no magic, it allows the controlled deployment of infrastructure in environment ready to do so and with team aware of the associated tooling. Fortunately, the recent Cloud concepts rise has eased the access to Infrastructure as Code with a generalization of the API to access any kind of cloud services. Also each cloud provider proposes its own set of tools to create Infrastructure template. AWS has CloudFormation and Azure has ARM Template.
On the private Cloud side, we can take the example of OpenStack with its set of API available for each of the OpenStack project.
However, while all those tools offer a wide range of possibilities, each of them is limited to its own platform, which implies that an IT guy would need to learn each of this tools syntaxes to be efficient on all the platforms.

1.2 Terraform from Hashicorp

Terraform is an Infrastructure as Code tool from Hashicorp. As other product in the area, it offers the capability to deploy infrastructure in different cloud environment by coding a template. The strenght of Terraform is its capabilities to address all the main Cloud Provider, with a similar syntax, thus enabling IT Ops to be efficient on different platforms through the same tool. A second strenght of Terraform is its hability to plan the infrastructure to be deployed. Before the deployment, the plan checks the infrastructure template and identifies the differents resource to be deployed. A state can be saved to ensure that what is coded will be what is deployed, in case of uncontrolled modification on the code.
Hashicorp provides ample documentation for Cloud deployment on its website. However, a good understanding of the Cloud platforms used remains a strong requirement.

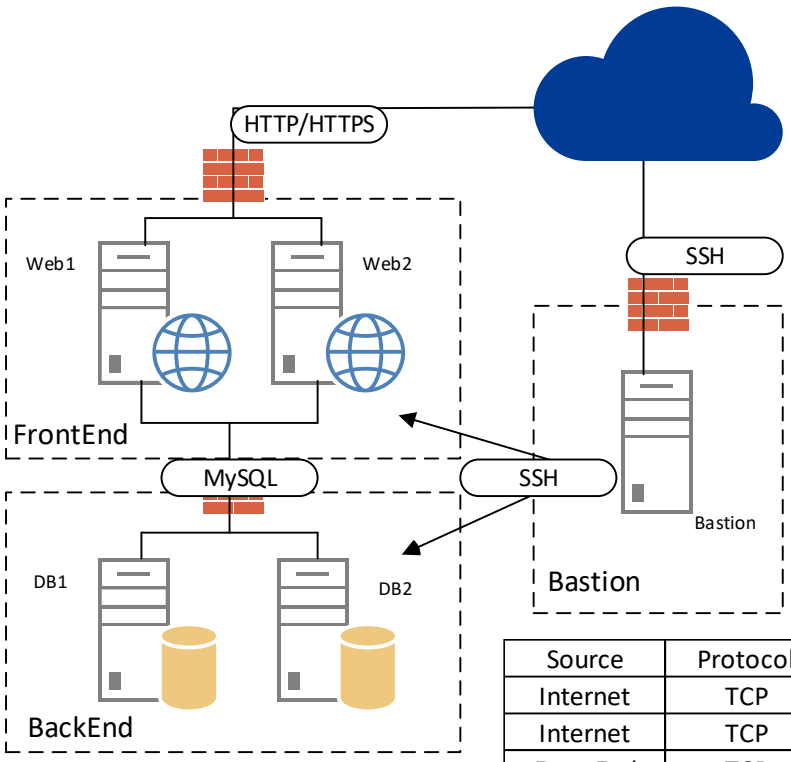
2 Target Architecture

To illustrate the capabilities of Terraform, we will describe a sample architecture and translate it to Azure equivalent services. We will then detail the Terraform code required for each of the Azure services to obtain the described architecture

2.1 Architecture description

The target architecture is a standard one, composed of Web Servers exposed to the internet and Back-End Database servers to store application data. A load balancer is placed in front onf the Web instances. An additional basion server is deployed to allow administration of the servers. All servers are Linux Servers. Network filtering is required to secure access to the servers. Only HTTP/HTTPS is allowed from the internet to the Web servers. Only the Database taffic is allowed from the Front-End to the Database Servers. Also SSH from the internet to the bastion is allowed and SSH from the bastion to all the other servers is allowed. A illustration of the proposed architecture is diplayed in the next chapter

2.2 Architecture Schema



Source	Protocol	Port	Target
Internet	TCP	80/443	FrontEnd
Internet	TCP	22	Bastion
FrontEnd	TCP	3306	BackEnd
Bastion	TCP	22	FrontEnd
Bastion	TCP	22	BackEnd

2.3 Related Azure Building Blocks

Now that we have a generic view of the architecture, let's translate it to Azure service.

2.3.1 Resource Group and Network

First thing first, in the Azure Resource Manager model, every resource is in a Resource Group. A resource group is as Microsoft describes:

> A container that holds related resources for an Azure solution. The resource group can include all the resources for the solution, or only those resources that you want to manage as a group. You decide how you want to allocate resources to resource groups based on what makes the most sense for your organization.

The second aspect is in the Network. In traditional environment, there is obviously a Network topology available on which the IT infrastructure relies. In Azure, the Network environment and also the Network isolation form the rest of the world is provided by an Azure Network (that we call a vNet). Microsoft describes more what are the features of the virtual Network here. For us, suffice to say that it allows us to isolate the resources and to declares what Network should access

to our architecture. The vNet will requires subnet range to be declared. We will take the range 10.0.0.0/20 in CIDR notation.

2.3.2 Subnets and Network Security Groups

After creation of the Azure vNet, Subnets have to be created to allow the use of the network capabilities. Subnet ranges are associated with each subnet. Obviously, the subnet IP range should be included in the range(s) declared at the vNet level. In our sample architecture, we have the following subnets:

- Front-End Subnet with the associated IP range 10.0.0.0/24
- Back-End Subnet with the associated IP range 10.0.1.0/24
- Bastion Subnet with the associated IP range 10.0.2.0/24

Without going in details here about the routing in Azure vNet, it is still interesting to note that all Azure Subnet are by default routable between each other and are able to access to the Internet, if the appropriate filtering rules on the security features are present.

Regarding the Network filtering, we use for the sample architecture the built-in Firewall in Azure which is the Network Security Rule. For our purpose, we have to understand the following:

- A Network Security Group provides Firewall features allowing the creation of Allow or Deny rules
- It is a Statefull Firewall in the sense that it automatically aloows the return traffic once a network flow is allowed
- It is a distributed firewall applying the rules at the Virtual NIC level (meaning the VMs NICs). However, it is possible to apply a Network Security Group on a Subnet. In this case, all NICs connected to this subnet will have the same set of rules applyed.
- Only one Network Security Group can be applyed to a NIC. Which means that in case of VMs requiring additional rules, a dedicated Network Security Group is required to be applied to those NICs.

In our sample architecture, to keep things simple, and since we do not have more requirement, we will create the following Network Security Group (NSG):

- NSG-Front-End, applyed to the Front-End Subnet, allowing the incoming HTTP/S traffic from the Internet to the Web Front-End VMs, through a load balancer. SSH traffic from the Bastion Subnet will also be allowed.
- NSG-Back-End, applyed to the Back-End subnet and allowing the Database traffic access from the Web Front-End instance and the SSH access from the Bastion subnet.
- NSG-Bastion, applyed to the Bastion Subnet and allowing only the SSH access from the Internet.

2.3.3 Azure Load Balancer

For the purpose of providing load balancing feature from the http service, we use the Azure Load Balancer feature. It consist of a load balancer managed by Azure and distributing the configured network traffic to the decalred virtual machines. Technically speaking, the following is requiring when implementing Azure Load Balancer service:

- An Azure Load Balancer, either Internet Facing or internal. In our case we create an Internet facing Load Balancer since we want to provide the feature for traffic coming from the Internet.
- A Front-End IP configuration, on which a public IP isassociated to allow the Internet Access
- A Back-End pool, which will allow the redirection of traffic from the Internet to the members of the pool. It can target either a single virtual machine or an availability set, wich we ill see later.
- An health probes which is used by the load balancing system to determine if the back-end pool members are avaialble or not
- Load balancing rules, on which we decalre the underlying mechanics of the load balancing, namely the protocol and port on the front end side, the back end port on the backend side, the health probe that was created earlier...
- Last, it is possible to implement, optionnaly, NAT rules to allow access on specific port to members of the bac-end pool through the public IP associated to the Azure Load Balancer.

2.3.4 Public IP

For acces from the Internet, since the use case of the Azure vNet is to provide isolation, public IP associated to the asset we want to give access to is requried. In the sample architecture, we have to End-Point accessible from the Internet:

- The Web Front-End through the load balancer
- The Bastion Server

Thus we need 2 public IP in the architecture. Those IP can either be dynamic or static. Depending of the use case, we could chose both. In any case, Azure will automaticlly create a public DNS hostname on which the assets will be accessible. However, for custom domain name, it culd be usefull / mandatory to use a fixed Public IP Address.

2.3.5 Virtual machine and storage

Last but not least, for the sample architecture, virtual servers are required. In Azure, those are simply called Azure VMs. As stated earlier, we will have:

- 2/3 front-end VMs with an Apache server for the web app role,
- back-end VMs with mariaDB server installed
- 1 bastion VM with Ansible installed for future configuration actions

In the sample architecture, no specifics compute or memory requirements exist. Small VMs can be used. However, to avoid issue with CPU credits, for now the F1 VMs will be used. This vm has the following characteristics

- 1vCPU
- 2GO of RAM
- Up to 2 NICs
- Compatible with Premium storage (in this case called F1S)

Regarding the Storage, Azure offers the managed disks features since the begining of 2017. It simplify greatly the storage aspect for the VMs, by hiding all the associated storage account underlying the storage aspects. For the sample architecture, all VMs will make use of managed disks. Additional Data disks will be created and associated with the VMs.

3 Coding the architecture with Terraform

At this point, the target architecture is now described in Azure services. The next step is the Infrastructure coding in Terraform.

3.1 Registering Terraform with an Azure Subscription

Before deploying with Terraform on an Azure subscription, Terraform first need to be installed. The installation is a simple process and the binaries are available for a wide range of Oses.
Once Terraform is installed, it needs to be registerd on the Azure Active Directory tenant associated to the subscription. A service Principal is required to provides Terraform with th enecessary credentials to provision resource on the Azure subscription. Microsoft provides documentation to register Terraform through the AZ CLI. For a more graphical experience, the configuration is very nicely detailed on Stanislas Quastana blog here.
Whatever the mean used to register the application in Azure, after the process is completed, the following infromation should be available:

- The Azure subscription ID
- The Azure Client ID
- The Azure Client Secret
- The Azure Tenant ID

Thise values should be declared in a .tf file as follow:

```
#################################################################
# This file contains all secret that are not to be exposed in Git
#################################################################



#################################################################
# Variables for Azure Registration
#################################################################

variable "AzureSubscriptionID" {
  type    = "string"
  default = ""
}

variable "AzureClientID" {
  type    = "string"
  default = ""
}

variable "AzureClientSecret" {
  type    = "string"
  default = "
```

```
}

variable "AzureTenantID" {
  type    = "string"
  default = ""
}
```

On this file, we declare variables to be used in the template, which will also be a .tf file. However, by splitting these sensitive information from the main template, we can avoid the exposition of the credentials used by Terraform, particularly when using a Git repository to store the template code.
On a syntax point of view, the variables are declared with the use of a variable block with the syntaxe:

```
variable "variable_name" {
  type    = "variable_type" #can be string, list or map
  default = "default_value_of_variable" #optional default value of the variable
}
```

In the main template file, the access to the subscription is done by using the following code:

```
#########################################################################
# Access to Azure
#########################################################################


# Configure the Microsoft Azure Provider with Azure provider variable defined in
AzureDFProvider.tf

provider "azurerm" {

    subscription_id = "${var.AzureSubscriptionID}"
    client_id       = "${var.AzureClientID}"
    client_secret   = "${var.AzureClientSecret}"
    tenant_id       = "${var.AzureTenantID}"
}
```

The provider azurerm is the part Terraform that talks to Azure API. Documentation on the available resource for this provider is available on Terraform site here.

3.2 Azure Building blocks in Terraform template

3.2.1 Terraform variables

As for all coding, Terraform makes use of variables. Again those variables are declared in a .tf file. While it is possible to declare everything in one .tf file only, it eases the management of the code for reuse to split again in another variable file the variables used by the template. On a security standpoint, I usually tend to store all my credentials/secret variable in one .tf file and all my template variable containing “insensitive” data in a variables.tf file. For the sample architecture, the additional “sensitive variables” are used:

- The VM admin name, declared as follow:

```
# Variable defining VM Admin Name

variable "VMAdminName" {

    type    = "string"
    default = "vmadmin"
}
```

- The VM admin password, declared as follow
  This password is not necessarily used if an SSH key is used instead

```
# Variable defining VM Admin password

variable "VMAdminPassword" {

    type    = "string"
    default = "P@ssw0rd!"
}
```

- The SSH public key used on the VMs

```
# Variable defining SSH Key
variable "AzurePublicSSHKey" {
  type    = "string"
  default = "ssh-rsa rsa-key-20170707"
}
```

The others variables used for the template are:

- The Azure Region, allowing to specify where in Azure the resources are deployed. The region is declared as a variable block as follow:

```
# Variable to define the Azure Region

variable "AzureRegion" {

    type    = "string"
    default = "westeurope"
}
```

For this scenario, the West Europe Region is chosen, with the string “westeurope”

- Tags are useful to classify resources by other means than their name. As example, we use a tag called TagEnvironment and another called TagUsage

```
# Variable to define the Tag

variable "TagEnvironment" {

    type    = "string"
    default = "BasicTemplateLinux"
}

variable "TagUsage" {

    type    = "string"
    default = "Lab"
}
```

- Last we use a variable to define the name of the resource group. While this name does not necessarily have to be variabilized, it eases the reuse of the code for later usage:

```
# Variable to define the Resource Group Name

variable "RSGName" {

    type    = "string"
    default = "RSG-BasicLinux"
}
```

3.2.2 Resource Group creation

After all the variables specifications, the template coding is possible. The first resource to code is the resource group. We already explain what was hidden behind this name in previous chapter. As for all resource in Terraform, we declare a

resource, in this case a resource called azurerm_resource_group. This resource requires a name, a location which is the Azure region, and optionally tags. The code looks as follow:

```
resource "azurerm_resource_group" "RSG-BasicLinux" {

    name        = "${var.RSGName}"
    location    = "${var.AzureRegion}"

    tags {
    environment = "${var.TagEnvironment}"
    usage       = "${var.TagUsage}"
    }
    }
```

3.2.3 Azure virtual network creation

As discussed earlier, the network isolation is relying on an Azure virtual network. This resource requires obviously a name, an associated resource group, which we created in the previous chapter, a location and a IP range. In Terraform, the resource is identified by the name azurerm_virtual_network. The code looks as follow:

```
resource "azurerm_virtual_network" "vNET-BasicLinux" {

    name = "vNET-BasicLinux"
    resource_group_name = "${azurerm_resource_group.RSG-BasicLinux.name}"
    address_space = ["10.0.0.0/20"]
    location = "${var.AzureRegion}"

    tags {
    environment = "${var.TagEnvironment}"
    usage       = "${var.TagUsage}"
    }
}
```

3.2.4 Subnet and Network Security Group creation

Next are the subnet and the associated Network Security Group. We declare the resource respectively with azurerm_subnet and azurerm_network_security_group. In terms of coding, we declare first the Network Security Group and we associate its id to the corresponding subnet.
The required parameters for the azurerm_network_security_group are the name, the location and the resource group. The id is provided by the Azure API and can be exported as an output by Terraform. The code for the Front-End Subnet Security Group is as follow:

```
resource "azurerm_network_security_group" "NSG-Subnet-BasicLinuxFrontEnd" {
```

```
    name = "NSG-Subnet-BasicLinuxFrontEnd"
    location = "${var.AzureRegion}"
    resource_group_name = "${azurerm_resource_group.RSG-BasicLinux.name}"

}
```

In Terraform, it is possible to declare the Security Group rules in the Network Security Group or independently. In this case, resource azurerm_network_security_rule is used. For the Front-End Subnet Network Security Group, we want to allow the HTTP/HTTPS traffic from the Internet to the Front-End Subnet. We also want to allow SSH from the Bastion Subnet. The code is as follow:

```
    ################################################
    # Rules Section
    ################################################

    # Rule for incoming HTTP from internet

resource "azurerm_network_security_rule" "AlltoFrontEnd-OK-HTTPIN" {

    name                        = "OK-http-Inbound"
    priority                    = 1100
    direction                   = "Inbound"
    access                      = "Allow"
    protocol                    = "TCP"
    source_port_range           = "*"
    destination_port_range      = "80"
    source_address_prefix       = "*"
    destination_address_prefix  = "10.0.0.0/24"
    resource_group_name         = "${azurerm_resource_group.RSG-BasicLinux.name}"
    network_security_group_name = "${azurerm_network_security_group.NSG-Subnet-
BasicLinuxFrontEnd.name}"

}

    # Rule for incoming HTTPS from internet

resource "azurerm_network_security_rule" "AlltoFrontEnd-OK-HTTPSIN" {

    name                        = "AlltoFrontEnd-OK-HTTPSIN"
    priority                    = 1101
    direction                   = "Inbound"
    access                      = "Allow"
    protocol                    = "TCP"
```

```
    source_port_range           = "*"
    destination_port_range      = "443"
    source_address_prefix       = "*"
    destination_address_prefix  = "10.0.0.0/24"
    resource_group_name         = "${azurerm_resource_group.RSG-BasicLinux.name}"
    network_security_group_name = "${azurerm_network_security_group.NSG-Subnet-
BasicLinuxFrontEnd.name}"

}


    # Rule for incoming SSH from Bastion

resource "azurerm_network_security_rule" "BastiontoFrontEnd-OK-SSHIN" {

    name                        = "BastiontoFrontEnd-OK-SSHIN"
    priority                    = 1102
    direction                   = "Inbound"
    access                      = "Allow"
    protocol                    = "TCP"
    source_port_range           = "*"
    destination_port_range      = "22"
    source_address_prefix       = "10.0.2.0/24"
    destination_address_prefix  = "10.0.0.0/24"
    resource_group_name         = "${azurerm_resource_group.RSG-BasicLinux.name}"
    network_security_group_name = "${azurerm_network_security_group.NSG-Subnet-
BasicLinuxFrontEnd.name}"

}


    #Rule for outbound to Internet traffic

resource "azurerm_network_security_rule" "FrontEndtoInternet-OK-All" {

    name                        = "FrontEndtoInternet-OK-All"
    priority                    = 1103
    direction                   = "Outbound"
    access                      = "Allow"
    protocol                    = "*"
    source_port_range           = "*"
    destination_port_range      = "*"
    source_address_prefix       = "10.0.0.0/24"
    destination_address_prefix  = "*"
    resource_group_name         = "${azurerm_resource_group.RSG-BasicLinux.name}"
```

```
    network_security_group_name = "${azurerm_network_security_group.NSG-Subnet-
BasicLinuxFrontEnd.name}"


}
```

As displayed, the rule declaration follows the logic of a classic firewall with the declaration of source / destination. Additionally, reference to the resource group and the network security group is required to associate the rule object with the appropriate security group.
It is interesting to note that while the Network Security Group resource and the subnet resource can be tagged, it is not the case for the security group rule. In effect this resource is totally included in the Security Group in Azure so it does not come as a big surprise.
In the code displayed, we can see that the source / target ip ranges are fixed values. Optimization of the code is possible by using variables for the subnets and the associated IP ranges.
Further details on the Network Security Rules is available in the template which is located here. To summarize, we want to allow SSH from Internet to the Bastion Subnet, incoming SSH from the Bastion subnet to all the other subnets and Database traffic from the FE to the BE. A last rule allowing all outbound traffic to the Internet is coded on each Security Groups.

3.2.5 Public IP Address

We defined earlier that 2 public IP are required for the sample architecture. One is aimed to allow access on the Web instance through the load balancer and the other is associated to the Bastion server. To code Azure public IP Address, we use the resource azurerm_public_ip. This resource has the following parameters:

- Name
- Location
- Resource group
- Public IP allocation
- Domain name label

We can also add the optional tags.
The domain name label is in fact a prefix on the DNS owned by Azure. In the final DNS name, we have the following structure:
<domain_name_label>.<azureregion>.cloudapp.azure.com

Since the name is hosted in the Azure associated namespace, it must be unique in the region. To ensure the uniqueness, Terraform allows the use of a random provider which in effect can create a random string. This random string can then be added to the domain name label parameter.
Thus in the code we have first the resource associated with the random prefix. This resource is called random_string. We specify the length and if we want to allow special characters, number and upper characters. In the case of a DNS name, we avoided all non-alphabetic character and upper characters. The length is equal to 5:

```
resource "random_string" "PublicIPfqdnprefixFE" {
```

```
    length = 5
    special = false
    upper = false
    number = false
}

resource "azurerm_public_ip" "PublicIP-FrontEndBasicLinux" {

    name                            = "PublicIP-FrontEndBasicLinux"
    location                        = "${var.AzureRegion}"
    resource_group_name             = "${azurerm_resource_group.RSG-
BasicLinux.name}"
    public_ip_address_allocation    = "static"
    domain_name_label               =
"${random_string.PublicIPfqdnprefixFE.result}dvtweb"

    tags {
    environment = "${var.TagEnvironment}"
    usage       = "${var.TagUsage}"
    }

}
```

The value created with the randam string is added to the domain name label with the interpolation ${random.string.<Random_String_Name>.result}.

We have a similar declaration for the bastion server:

```
resource "random_string" "PublicIPfqdnprefixBastion" {


    length = 5
    special = false
    upper = false
    number = false
}

resource "azurerm_public_ip" "PublicIP-BastionBasicLinux" {

    name                            = "PublicIP-BastionBasicLinux"
    location                        = "${var.AzureRegion}"
```

```
    resource_group_name             = "${azurerm_resource_group.RSG-
BasicLinux.name}"
    public_ip_address_allocation    = "static"
    domain_name_label               =
"${random_string.PublicIPfqdnprefixBastion.result}dvtbastion"

    tags {
    environment = "${var.TagEnvironment}"
    usage       = "${var.TagUsage}"
    }
}
```

3.2.6 Azure Load Balancer

We listed earlier the elements composing a load balancer:

- An Internet facing Azure Load Balancer
- A Front-End IP configuration
- A Back-End pool
- Load balancing rules
- A health probe

The corresponding object to code in Terraform are:

- azurerm_lb
- azurerm_lb_backend_address_pool
- azurerm_lb_rule
- azurerm_lb_probe

The front-end IP configuration is linked to the azurerm_lb object. This object takes the following parameters:

- Name
- Location
- Resource group name
- Front-End IP configuration

The front end IP configuration is defined in a dedicated block. It takes also a set of parameters:

- Name
- Public IP address

To be noted, in the sample architecture, the front end config takes a public IP parameter because it is a public facing load balancer. In case of an internal load balancer, other parameters are used and detailed in the Terraform documentation. The code for this part of the Azure load balancer is displayed below

```
resource "azurerm_lb" "LB-WebFrontEndBasicLinux" {

    name                        = "LB-WebFrontEndBasicLinux"
    location                    = "${var.AzureRegion}"
    resource_group_name         = "${azurerm_resource_group.RSG-BasicLinux.name}"

    frontend_ip_configuration {

        name                    = "weblbbasiclinux"
        public_ip_address_id    = "${azurerm_public_ip.PublicIP-
FrontEndBasicLinux.id}"
    }

    tags {
    environment = "${var.TagEnvironment}"
    usage       = "${var.TagUsage}"
    }
}
```

The second object to be coded is the azurerm_lb_backen_address_pool. This object is equivalent to the backend pool found in the Azure Web portal. It takes as parameters the following:

- Name
- Resource group name
- Load balancer id

While the first parameters are self-explanatory, the last one is to attach the object to the appropriate load balancer. Another interesting point is that contrary at what is available on the Azure portal, we do not attach here the backend pool to an availability set or any VM. As displayed in the code below, no reference to the backend virtual machine behind the load balancer is visible.

```
resource "azurerm_lb_backend_address_pool" "LB-WebFRontEndBackEndPool" {

    name                = "LB-WebFRontEndBackEndPool"
    resource_group_name = "${azurerm_resource_group.RSG-BasicLinux.name}"
    loadbalancer_id     = "${azurerm_lb.LB-WebFrontEndBasicLinux.id}"

}
```

The 3rd object is the azurerm_lb_probe which is equivalent to the health probe, as expected. It takes as parameters the following:

- Name
- Resource group name
- Load balancer id

- Port

The port parameter allows us to specify on which port the load balancer is processing the health probe on the Back-End pool. The code is as follow:

```
resource "azurerm_lb_probe" "LB-WebFrontEnd-httpprobe" {

    name                = "LB-WebFrontEnd-httpprobe"
    resource_group_name = "${azurerm_resource_group.RSG-BasicLinux.name}"
    loadbalancer_id     = "${azurerm_lb.LB-WebFrontEndBasicLinux.id}"
    port                = 80


}
```

Last, we code the load balancer rule. This resource takes more parameters. It references the load balancer by its id, the probe by its id and also the front end configuration part and the backend configuration part.

```
resource "azurerm_lb_rule" "LB-WebFrondEndrule" {

    name                           = "LB-WebFrondEndrule"
    resource_group_name            = "${azurerm_resource_group.RSG-
BasicLinux.name}"
    loadbalancer_id                = "${azurerm_lb.LB-WebFrontEndBasicLinux.id}"
    protocol                       = "tcp"
    probe_id                       = "${azurerm_lb_probe.LB-WebFrontEnd-
httpprobe.id}"
    frontend_port                  = 80
    frontend_ip_configuration_name = "weblbbasiclinux"
    backend_port                   = 80
    backend_address_pool_id        = "${azurerm_lb_backend_address_pool.LB-
WebFRontEndBackEndPool.id}"


}
```

At this point, we still have no reference to the associated VMs in the back-end pool. In fact, we will define this dependency when configuring the VM's NIC.

3.2.7 Data disks

Until now, few details were given on the storage aspects. This is due to the use of the managed disk functionality. Before the managed disk GA, it was necessary to manage the storage for VMs with storage account. In this sample architecture, Managed Disks are used for the data disks and the OS disks. The OS Disks are created at the same time as the VMs while the data disks are created beforehand and attached to the VMs in the VMs configuration.
The corresponding resource in Terraform is called azurerm_managed_disk. It takes the following parameters:

- Name
- Location
- Resource group name
- Storage account type
- Create option
- Disk size in gb

We also use here the parameter count which allow the creation of the objects, in this case the data disks but it can be used for others resources, in multiple number corresponding to the count value. The storage account type parameter allows to specify the storage tier associated to the managed disk. It can be either standard_lrs, indicating a locally redundant storage standard or premium _lrs, indicating locally redundant storage premium.
The code for the managed disks creation to be associated to the front-end servers is as follow:

```
resource "azurerm_managed_disk" "WebFrontEndManagedDisk" {

    count                   = 3
    name                    = "WebFrontEnd-${count.index + 1}-Datadisk"
    location                = "${var.AzureRegion}"
    resource_group_name     = "${azurerm_resource_group.RSG-BasicLinux.name}"
    storage_account_type    = "Standard_LRS"
    create_option           = "Empty"
    disk_size_gb            = "127"

    tags {
    environment = "${var.TagEnvironment}"
    usage       = "${var.TagUsage}"
  }

}
```

The size is impacting with the pricing of the storage for the VMs.
Also for the name parameter, interpolation with the count index value is done by adding in the name's string the value ${count.index+1}. The + 1 allows to number the data dis starting from 1 instead of 0 which is the first value of the index.
We create data disks for each group of servers, meaning the front-end servers, the back-end servers and the bastion server.

3.2.8 VMs creation

The VMs are associated to a NIC. This object is distinct from the VM object itself and is thus created separately. NICs are created first, and then are the VMs created.

3.2.8.1 NICs Creation

The NIC object in Terraform is called azurerm_network_interface. It takes the following parameters:

- Name
- Location
- Resource group name
- Ip configuration

It also takes a count parameter, which allows the creation of multiple instance of the object.
The ip configuration is a block code, taking the following parameters:

- Name
- Subnet id
- Private ip address allocation
- Load balancer backend address pool id

This last parameter is what allows the association to the load balancer and the appropriate backend pool described earlier.
The code is as follow:

```
resource "azurerm_network_interface" "WebFrontEndNIC" {

    count                     = 3
    name                      = "WebFrontEnd${count.index +1}-NIC"
    location                  = "${var.AzureRegion}"
    resource_group_name       = "${azurerm_resource_group.RSG-BasicLinux.name}"

    ip_configuration {

        name                                          = "ConfigIP-NIC${count.index +
1}-WebFrontEnd${count.index + 1}"
        subnet_id                                     = "${azurerm_subnet.Subnet-
BasicLinuxFrontEnd.id}"
        private_ip_address_allocation                 = "dynamic"
        load_balancer_backend_address_pools_ids       =
["${azurerm_lb_backend_address_pool.LB-WebFRontEndBackEndPool.id}"]
    }

    tags {
        environment = "${var.TagEnvironment}"
        usage       = "${var.TagUsage}"
    }
```

```
}
```

3.2.8.2 Availability set creation

Once the NICs and the data disks are created, all required resource for the VMs are present. The provisioning of the VMS can be performed. However, to enable the Azure SLA related to the VMs, the availability set is required. This resource is simply a logical group of VMs, allowing the multiple associated VMs to be updated and patched independently. VMs in availability set are associated with a SLA of 99,95 % while stand-alone VMs are associated with only 99,9 %.
The resource in Terraform is called azurerm_availability_set and takes the following parameters:

- Name
- Location
- Resource group name

Additional parameters to specify the associated update domain and fault domain can be specified. If not, the default value applies, 5 for the update domains, 3 for the fault domains.

```
resource "azurerm_availability_set" "BasicLinuxWebFrontEnd-AS" {

    name                    = "BasicLinuxWebFrontEnd-AS"
    location                = "${var.AzureRegion}"
    managed                 = "true"
    resource_group_name     = "${azurerm_resource_group.RSG-BasicLinux.name}"
    tags {
        environment = "${var.TagEnvironment}"
        usage       = "${var.TagUsage}"
  }
}
```

3.2.8.3 VMs creation

The VM object used for Terraform is called azurerm_virtual_machine. It relies on the previously created object and requires the following parameters:

- Count
- Name
- Location
- Resource group name
- Network interface ids
- Vm size
- Availability set id

Optionally, it is possible to specify a dependency on a particular resource with the parameters depends_on. In this case, the resources are specified as a list between square brackets [ ].

```
resource "azurerm_virtual_machine" "BasicLinuxWebFrontEndVM" {

    count                   = 3
    name                    = "BasicLinuxWebFrontEnd${count.index +1}"
    location                = "${var.AzureRegion}"
    resource_group_name     = "${azurerm_resource_group.RSG-BasicLinux.name}"
    network_interface_ids   =
["${element(azurerm_network_interface.WebFrontEndNIC.*.id, count.index)}"]
    vm_size                 = "${var.VMSize}"
    availability_set_id     = "${azurerm_availability_set.BasicLinuxWebFrontEnd-
AS.id}"
    depends_on              = ["azurerm_network_interface.WebFrontEndNIC"]
```

In this portion of code, the element() function is used to associate to the current virtual machine object in the count the equivalent object NIC created previously. Element() is a Terraform built-in function used to read element of a variable list. When the count parameter is used in an object, tis object becomes a list containing the same number of object as the count value. The element() function becomes necessary hen the association with one element of the list is required in another resource creation with the count parameter used.

The storage image reference is defined as a dedicated code block. This block allows to identify which VM image is chosen and has the following parameters:

- Publisher
- Offer
- Sku
- Version

```
    storage_image_reference {

        publisher   = "${var.OSPublisher}"
        offer       = "${var.OSOffer}"
        sku         = "${var.OSsku}"
        version     = "${var.OSversion}"

    }
```

The storage for the OS disk is also in a dedicated code block called storage os disk and requires the following parameters:

- Name
- Caching
- Create option

- Managed disk type

```
    storage_os_disk {

        name                = "WebFrontEnd-${count.index + 1}-OSDisk"
        caching             = "ReadWrite"
        create_option       = "FromImage"
        managed_disk_type   = "Standard_LRS"

    }
```

The storage for the data disk requires a dedicated code block called storage_data_disk and requires the following parameters:

- Name
- Managed disk id
- Create option
- Lun
- Disk size in gb

```
    storage_data_disk {

        name                =
"${element(azurerm_managed_disk.WebFrontEndManagedDisk.*.name, count.index)}"
        managed_disk_id     =
"${element(azurerm_managed_disk.WebFrontEndManagedDisk.*.id, count.index)}"
        create_option       = "Attach"
        lun                 = 0
        disk_size_gb        =
"${element(azurerm_managed_disk.WebFrontEndManagedDisk.*.disk_size_gb,
count.index)}"

    }
```

In this portion of code, the element() function is used again to relate the data disk list resource with the corresponding element of the VM resource list.

Os profile is also configured in a dedicated code block which takes the following parameters:

- Computer name
- Admin username
- Admin password

```
    os_profile {

        computer_name   = "WebFrontEnd${count.index + 1}"
        admin_username  = "${var.VMAdminName}"
        admin_password  = "${var.VMAdminPassword}"

    }
```

Last, depending of the VM's OS, an additional code block is required. In this sample architecture, since VM's OS is Linux, the required code block is os profile linux config. It takes the following parameters:

- Disable password authentication
- Ssh key which is also a code block and allows to specify the path for the ssh key and the ssh public key

```
    os_profile_linux_config {

    disable_password_authentication = true

    ssh_keys {
      path     = "/home/${var.VMAdminName}/.ssh/authorized_keys"
      key_data = "${var.AzurePublicSSHKey}"
    }
```

3.2.8.4 VM extension

The correspondence with the Azure VM extension is realized through the use of the Terraform resource azurerm_virtual_machine_extension. To use this resource, it is required to identify, as in the VM image, the publisher, the type and the version of this agent. The resource requires the following parameters

- Count
- Name
- Location
- Resource group name
- Virtual machine name
- Publisher
- Type
- Type handler version

Additionally, a JSO block code is used if particular settings are required for the agent. The code for the bastion VM's Linux custom script extension is displayed below:

```
resource "azurerm_virtual_machine_extension" "CustomExtension-basicLinuxBastion" {
```

```
  count                = 1
  name                 = "CustomExtensionBastion-${count.index +1}"
  location             = "${var.AzureRegion}"
  resource_group_name  = "${azurerm_resource_group.RSG-BasicLinux.name}"
  virtual_machine_name = "BasicLinuxBastion${count.index +1}"
  publisher            = "Microsoft.OSTCExtensions"
  type                 = "CustomScriptForLinux"
  type_handler_version = "1.5"
  depends_on           = ["azurerm_virtual_machine.BasicLinuxBastionVM"]

      settings = <<SETTINGS
        {
        "fileUris": [ "https://raw.githubusercontent.com/dfrappart/Terra-
AZBasiclinux/master/deployansible.sh" ],
        "commandToExecute": "bash deployansible.sh"
        }
SETTINGS

  tags {
    environment = "${var.TagEnvironment}"
    usage       = "${var.TagUsage}"
  }
}


# Creating virtual machine extension for FrontEnd

resource "azurerm_virtual_machine_extension" "CustomExtension-basicLinuxFrontEnd" {

  count                = 3
  name                 = "CustomExtensionFrontEnd-${count.index +1}"
  location             = "${var.AzureRegion}"
  resource_group_name  = "${azurerm_resource_group.RSG-BasicLinux.name}"
  virtual_machine_name = "BasicLinuxWebFrontEnd${count.index +1}"
  publisher            = "Microsoft.OSTCExtensions"
  type                 = "CustomScriptForLinux"
  type_handler_version = "1.5"
  depends_on           = ["azurerm_virtual_machine.BasicLinuxWebFrontEndVM"]

      settings = <<SETTINGS
        {
        "fileUris": [ "https://raw.githubusercontent.com/dfrappart/Terra-
AZBasiclinux/master/installapache.sh" ],
        "commandToExecute": "bash installapache.sh"
```

```
        }
SETTINGS

  tags {
    environment = "${var.TagEnvironment}"
    usage       = "${var.TagUsage}"
  }
}
```

4 Conclusion

In this article, we described a sample architecture, its translation to Azure building blocks and how to code the deployment with Terraform.
At this point, only basic features of Terraform are used. The template is not taking advantages of the DRY capabilities of Terraform with the use of Modules. Also, except for the element () function, no advances features are used.
We did not detail either the outpout capabilities of Terraform. However, some output are coded in the template which is available on GitHub here.
In another article, we will describe how to implement the same architecture sample while taking advantages of the Terraform modules.

Another Tech blog

My thoughts and experiences on Cloud related tek