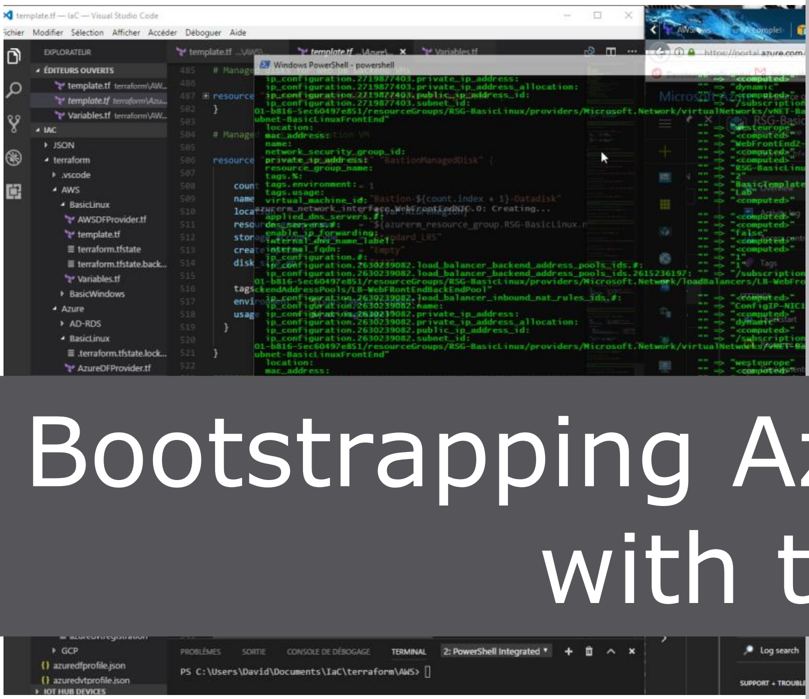


Teknews.cloud

Bootstrapping Azure VMs with terraform

David Frappart

24/04/2018

Contributor

Name	Title	Email
David Frappart	Cloud Architect	david@teknews.cloud

Version Control

Version	Date	State
1.0	2018/04/24	Final

Table of Contents

1 The initial config of VMs on Azure

In previous article, we used, without going in too much details, the VM Agents available on Azure for the initial config of the VM created through Terraform. In this article, i propose that we take some time to look at what we have available, and some limitation built in, with the agent and other Azure / Terraform stuff.

2 Options for bootstrapping Azure VMs

For those of you guys who have some knowledge of AWS, here comes the warning: There is no such thing as the ***userdata*** parameter for Azure VMs. Meaning, we cannot simply add some command behind an equivalent parameter and expect the platform to bootstrap the VMs. Or can we?
Actually, Microsoft provides on its platform many different agents, that can be used for configuration. The simplest one being the well-known custom extension script agent, followed by agent like the DSC (understand the PowerShell DSC) Agent, or Puppet and Chef Agent. We will limit our experimentation to the custom script in this article. We will also have a look at the ***custom_data*** parameter, exposed on the API and thus through Terraform, and check what we can do about it.

2.1 The custom extension script agent

The custom script extension aimed to provide a simple and easy way to execute script on VM at the 1st boot time.
If we take a look at the Microsoft documentation here for Windows and here for Linux, we can have a look on what the agent expects. Without surprise it is JSON based. Two configuration can be provided, a public and a protected one. For the public configuration:

- **fileUris**, for which we need to provide an http based uri pointing to the script file, for example, a github repo or an Azure storage account
- **commandToExecute** for which we will simply specify the command to execute. For example, if we got a bootscript.sh from the uri, we will have a command like bash bootscript.sh

```
{
  "fileUris": ["<url>"],
  "commandToExecute": "<command-to-execute>"
}
```

For the protected configuration, we have the following parameters:

- commandToExecute
- storageAccountName
- storageAccountKey

```
{
  "commandToExecute": "<command-to-execute>",
  "storageAccountName": "<storage-account-name>",
  "storageAccountKey": "<storage-account-key>"
}
```

The parameters are case sensitive, but since we will use Terraform as a medium between the API parameters and our code, it will not be a major problem.

2.2 The custom data parameter and the cloud-init script

Another way to bootstrap Azure VMs is through the use of Cloud-init scripts. Microsoft describes the terms of support for this solution and the VMs images on which we can use it:

Publisher	Offer	SKU	Version	cloud-init ready
Canonical	UbuntuServer	16.04-LTS	latest	yes
Canonical	UbuntuServer	14.04.5-LTS	latest	yes
CoreOS	CoreOS	Stable	latest	yes
OpenLogic	CentOS	7-CI	latest	preview
RedHat	RHEL	7-RAW-CI	latest	preview

To push a script through to an Azure VM, we use the custom_data parameter. This will only work on VM images in the list shown here. We will have a look at how we do that with terraform and since we are players, what happens when we use the parameter with Windows images.

3 Experimenting with the bootstrapping options with Terraform

3.1 The working environment

The working environment will be composed of an Azure Vnet with subnets, NSGs and VMs. Really, here what is important is to look into the agent configuration through Terraform. For an example config, i have published on Github a dedicated repo.

3.2 Deploying with the custom extension script and using Terraform templates

In a previous article, i used the following syntaxe for the custom script extension:

```
# Creating virtual machine extension for Backend

resource "azurerm_virtual_machine_extension" "CustomExtension-basicLinuxBackEnd" {
  count                  = 2
  name                   = "CustomExtensionBackEnd-${count.index +1}"
  location               = "${var.AzureRegion}"
  resource_group_name    = "${azurerm_resource_group.RSG-BasicLinux.name}"
  virtual_machine_name = "BasicLinuxDBBackEnd${count.index +1}"
  publisher              = "Microsoft.OSTCExtensions"
  type                   = "CustomScriptForLinux"
  type_handler_version = "1.5"
  depends_on             = ["azurerm_virtual_machine.BasicLinuxDBBackEndVM"]

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

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

First thing first, the depends_on parameter is really not necessary, since Terraform is quite smart and can manage the dependancies between resources.
In this extract of code, the interesting fact is that finally, we can use the same syntaxe for any Azure Agent, as long as we can get the publisher, type and type_handler_version. So in theory, we could swith the agent just by changing those parameters.
However, in a Terraform Module view, we are limited by the JSON parameters between the <<SETTINGS {} SETTINGS.
So out of the box, we need a different module for each different script that we want to push on the config. This is not really DRY in a sense that the same module is duplicated with simply a different JSON string.
Secondly, we may not have always access to an http(s) repo to store the script files.

Solving the second point is easy and is actually displayed on Microsoft documentation. It is possible to provide only the command to execute, which may be something like that:

```
"commandToExecute": "yum install -y epel-release > /dev/null && yum install -y
nginx > /dev/null && systemctl start nginx && echo 'bootscript done' >
/tmp/result.txt"
```

The first point can be solved through the use of Terraform template. Clearly here we are addressing an issue more on the IaC tool rather than on Azure, but nevertheless, it is interesting to look into it.

Terraform templates allow to reference long strings such as the JSON string in some of Azure Resources as a template file (or directly inline, but in this article, we will focus on the first case). The use of those template files are described in the Terraform documentation, but are not necessarily clear.
So let’s see.
Below is a example of a module for a custom script extension:

```
##################################################################################
#This module allows the creation of a CustomLinuxExtension and use a template for
#the JSON part
###################################################################################

#Variable declaration for Module

variable "AgentCount" {
  type    = "string"
  default = 1
}

variable "AgentName" {
  type = "string"
}

variable "AgentLocation" {
  type = "string"
}

variable "AgentRG" {
  type = "string"
}

variable "VMName" {
  type = "list"
}

variable "EnvironmentTag" {
  type = "string"
}

variable "EnvironmentUsageTag" {
  type = "string"
}

variable "AgentPublisher" {
  type    = "string"
  default = "Microsoft.Azure.Extensions" #for Linux custom extension

  #default = "microsoft.compute" #For Windows custom extension
}

variable "AgentType" {
  type    = "string"
  default = "CustomScript" #default value for Linux Agent

  #default = "customscriptextension" #Default value for Windos Agent
}

variable "Agentversion" {
  type    = "string"
  default = "2.0"
}

#Variable passing the string rendered template to the settings parameter
variable "SettingsTemplatePath" {
  type = "string"
}

#Resource Creation

data "template_file" "customscripttemplate" {
  template = "${file("${path.root}${var.SettingsTemplatePath}")}"
}

resource "azurerm_virtual_machine_extension" "Terra-CustomScriptLinuxAgent" {
  count                = "${var.AgentCount}"
  name                 = "${var.AgentName}${count.index+1}"
  location             = "${var.AgentLocation}"
  resource_group_name  = "${var.AgentRG}"
  virtual_machine_name = "${element(var.VMName,count.index)}"
  publisher            = "${var.AgentPublisher}"
  type                 = "${var.AgentType}"
  type_handler_version = "${var.Agentversion}"

  settings = "${data.template_file.customscripttemplate.rendered}"

  /*
                          settings = <<SETTINGS
                                {
                                        "commandToExecute": "yum install -y epel-
release nginx > /dev/null"
                                }
                        SETTINGS
                        */
  tags {
    environment = "${var.EnvironmentTag}"
    usage       = "${var.EnvironmentUsageTag}"
  }
}

#Module Output

output "RGName" {
  value = "${var.AgentRG}"
}
```

Let’s focus on the resource creation part. We declare 2 differents resource. The classic one i would say is the Azure resource targeting to deploy the Azure VM Agent. As discussed earlier, we can create nearly any Azure Agent with the appropriates values for the variables.

```
resource "azurerm_virtual_machine_extension" "Terra-CustomScriptAgent" {}
```

The other resource created is declared using a data template file resource.

```
data "template_file" "customscripttemplate" {
  template = "${file("${path.root}${var.SettingsTemplatePath}")}"
}
```

In the example, only one parameter is required. This parameter is the path of the template file. The use of the path.root variable allow us to indicate that the path provided should be from the root module perpective. For example, if we reference the ./Templates/Templatefile.tpl, Terraform will check the path form the location of the root module. Then, instead of using the balises with <<SETTINGS {} SETTINGS, we use the template with the .rendered attribute:

```
settings = "${data.template_file.customscripttemplate.rendered}"
```

And that’s it. We eliminate the JSON from our code and we can have a more reusable custom script module, since we do not need to specify the command in a JSON file for which the interpolation is not working.

3.3 Deploying with the custom data and the cloud-init script

3.3.1 The custom data parameter for Linux supported VMs

The custom data parameter has been around since about 2014 in Azure. We displayed earlier which Azure VM Images are supported with the cloud-init script and can take benefit of this parameter to push a script directly through the VM code instead of using a custom extension script.
The advantage of this approach are:

- No HTTP repo for the script file required, we can push the script directly
- Since it is included in the VM deployment code, there is no need for a custom script agent and thus the deployment requires one less resource per VM and is faster.

Below is the code for the VM deployment module with the use of the custom data parameter:

```
###################################################################################
#This module allows the creation of n Linux VM with 1 NIC
###################################################################################

#Variable declaration for Module

#The VM count
variable "VMCount" {
  type    = "string"
  default = 1
}

#The VM name
variable "VMName" {
  type = "string"
}

#The VM location
variable "VMLocation" {
  type = "string"
}

#The RG in which the VMs are located
variable "VMRG" {
  type = "string"
}

#The NIC to associate to the VM
variable "VMNICid" {
  type = "list"
}
```

```
#The VM size
variable "VMSize" {
  type    = "string"
  default = "Standard_F1"
}

#The Availability set reference

variable "ASID" {
  type = "string"
}

#The Managed Disk Storage tier

variable "VMStorageTier" {
  type    = "string"
  default = "Premium_LRS"
}

#The VM Admin Name

variable "VMAdminName" {
  type    = "string"
  default = "VMAdmin"
}

#The VM Admin Password

variable "VMAdminPassword" {
  type = "string"
}

# Managed Data Disk reference

variable "DataDiskId" {
  type = "list"
}

# Managed Data Disk Name

variable "DataDiskName" {
  type = "list"
}
```

```
# Managed Data Disk size

variable "DataDiskSize" {
  type = "list"
}

# VM images info
#get appropriate image info with the following command
#Get-AzureRMVMImagePublisher -location WestEurope
#Get-AzureRMVMImageOffer -location WestEurope -PublisherName <PublisherName>
#Get-AzureRmVMImageSku -Location westeurope -Offer <OfferName> -PublisherName
<PublisherName>

variable "VMPublisherName" {
  type = "string"
}

variable "VMOffer" {
  type = "string"
}

variable "VMsku" {
  type = "string"
}

#The boot diagnostic storage uri

variable "DiagnosticDiskURI" {
  type = "string"
}

#The Cloud-init script path

variable "CloudinitscriptPath" {
  type = "string"
}

variable "PublicSSHKey" {
  type = "string"
}

#Tag info
```

```
variable "EnvironmentTag" {
  type    = "string"
  default = "Poc"
}

variable "EnvironmentUsageTag" {
  type    = "string"
  default = "Poc usage only"
}

data "template_file" "cloudconfig" {
  #template = "${file("./script.sh")}"
  template = "${file("${path.root}${var.CloudinitscriptPath}")}"
}

#https://www.terraform.io/docs/providers/template/d/cloudinit_config.html
data "template_cloudinit_config" "config" {
  gzip          = true
  base64_encode = true

  part {
    content = "${data.template_file.cloudconfig.rendered}"
  }
}

#VM Creation

resource "azurerm_virtual_machine" "TerraVMwithCount" {
  count                 = "${var.VMCount}"
  name                  = "${var.VMName}${count.index+1}"
  location              = "${var.VMLocation}"
  resource_group_name   = "${var.VMRG}"
  network_interface_ids = ["${element(var.VMNICid, count.index)}"]
  vm_size               = "${var.VMSize}"
  availability_set_id   = "${var.ASID}"

  boot_diagnostics {
    enabled     = "true"
    storage_uri = "${var.DiagnosticDiskURI}"
  }

  storage_image_reference {
    #get appropriate image info with the following command
    #Get-AzureRmVMImageSku -Location westeurope -Offer windowsserver -PublisherName
microsoftwindowsserver
    publisher = "${var.VMPublisherName}"

    offer   = "${var.VMOffer}"
    sku     = "${var.VMsku}"
    version = "latest"
  }

  storage_os_disk {
    name              = "${var.VMName}${count.index+1}-OSDisk"
    caching           = "ReadWrite"
    create_option     = "FromImage"
    managed_disk_type = "${var.VMStorageTier}"
  }

  storage_data_disk {
    name            = "${element(var.DataDiskName,count.index)}"
    managed_disk_id = "${element(var.DataDiskId,count.index)}"
    create_option   = "Attach"
    lun             = 0
    disk_size_gb    = "${element(var.DataDiskSize,count.index)}"
  }

  os_profile {
    computer_name  = "${var.VMName}"
    admin_username = "${var.VMAdminName}"
    admin_password = "${var.VMAdminPassword}"
    custom_data    = "${data.template_cloudinit_config.config.rendered}"
  }

  os_profile_linux_config {
    disable_password_authentication = true

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

  tags {
    environment = "${var.EnvironmentTag}"
    usage       = "${var.EnvironmentUsageTag}"
  }
```

```
}

output "Name" {
  value = ["${azurerm_virtual_machine.TerraVMwithCount.*.name}"]
}

output "Id" {
  value = ["${azurerm_virtual_machine.TerraVMwithCount.*.id}"]
}

output "RGName" {
  value = "${var.VMRG}"
}
```

In the os profile block, we specify the reference to the cloud-init script:

```
  os_profile {
    computer_name  = "${var.VMName}"
    admin_username = "${var.VMAdminName}"
    admin_password = "${var.VMAdminPassword}"
    custom_data    = "${data.template_cloudinit_config.config.rendered}"
  }
```

As we can see, the value is referencing a data resource which is somehow similar to a template, except it is specific to the cloud-init config. We make this happen in two step.
The first step is referencing the script path with a variable:

```
#The Cloud-init script path

variable "CloudinitscriptPath" {
  type = "string"
}
```

This variable is used in the template_file data source:

```
data "template_file" "cloudconfig" {

  template = "${file("${path.root}${var.CloudinitscriptPath}")}"
}
```

We call the module with the value as follow to reference the script path:

```
CloudinitscriptPath = "./Scripts/installnginxubuntu.sh"
```

Again we are using the path.root variable to indicate that the path provided is from the root module perspective.
We then use the data source template_cloudinit_config with the gzip parameter and the base64_encode parameter to put the file in the correct format. Last, we use the part block to render the script file.

```
data "template_cloudinit_config" "config" {
  gzip          = true
  base64_encode = true

  part {
    content = "${data.template_file.cloudconfig.rendered}"
  }
}
```

The custom_data then references this last data source:

```
custom_data     = "${data.template_cloudinit_config.config.rendered}"
```

And with that we are able to push a script file on a supported linux VM and install for example NGINX with a source script like this one:

```
#!/bin/bash

echo "bootscript initiated" > /tmp/results.txt
sudo apt-get update -y
sudo apt-get install -y nginx
sudo systemctl start nginx


echo "bootscript done" >> /tmp/results.txt

exit 0
```

3.3.2 About the custom data for other VMs

For Windows VM, when we use the custom data, Terraform documentation indicates that the script file referenced will be copied to the VM.
Indeed, after deploying the VM with the custom data value filled with a reference to a script file, we can find in the C:\AzureData a file named CustomData.bin.
In the module we declare a variable for the script path:

```
variable "CloudinitscriptPath" {
  type = "string"
}
```

This path is then referenced in the custom_data

```
  os_profile {
    computer_name  = "${var.VMName}${count.index+1}"
    admin_username = "${var.VMAdminName}"
    admin_password = "${var.VMAdminPassword}"
    custom_data    = "${file("${path.root}${var.CloudinitscriptPath}")}"
  }
```

When calling the module we provide the following value:

```
  CloudinitscriptPath = "./Scripts/example.ps1"
```

So two things worthy of note here. First, we do not use the data template_file neither the data template_cloudinit_config. We are simply directly referencing the path in the custom_data.
Unfortunately, that's about all and the script file is only copied, not executed. To work around this limitation, just for the technical challenge, we can add a custom script extension with a template file in which we are making the necessary adjustments to change the file to a script format and then run the file:

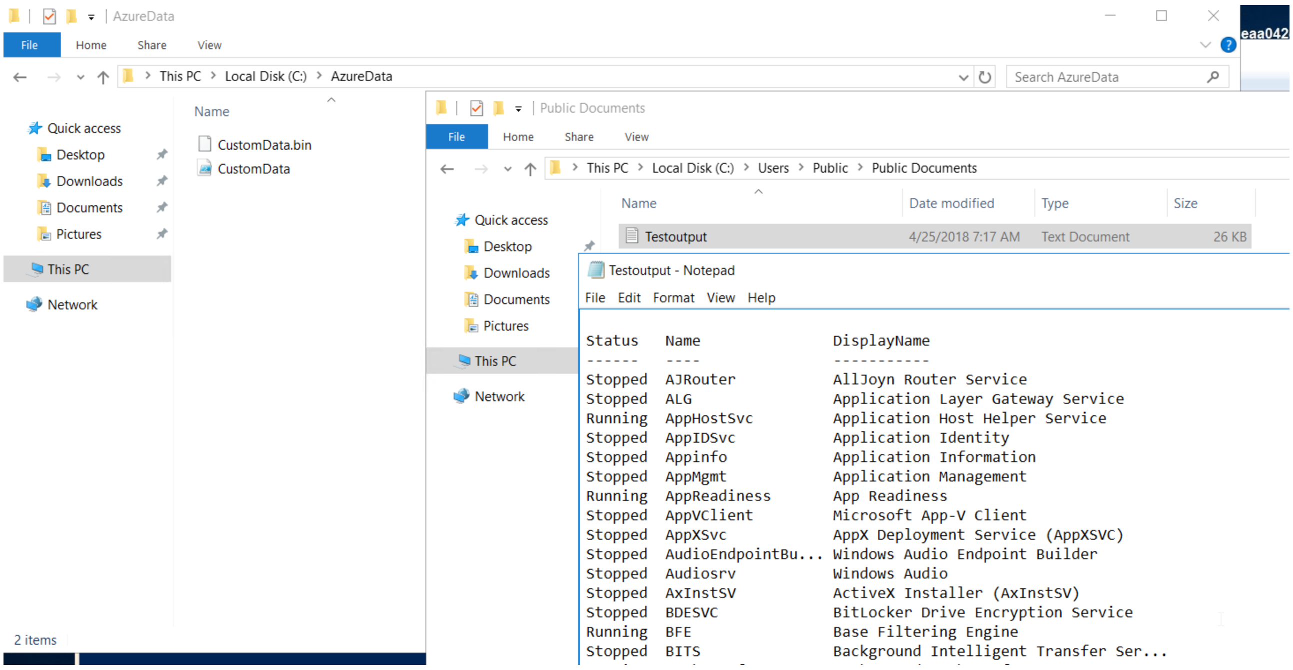
```
{
    "commandToExecute": "powershell -command install-windowsfeature web-
server;copy-item \"c:\\AzureData\\CustomData.bin\"
\"c:\\AzureData\\CustomData.ps1\";\"c:\\AzureData\\CustomData.ps1\""

}
```

Then, with the same module for the custom script extension as earlier, we can make the agent run the script file provided through the custom data even if the Image is a Windows one. An important point to be noted here, we have to escape the " and the \ as those are characters that needs to be escaped in the JSON format. Failure to doing so will result in an error when planning the terraform config.
To be sure of our test, we specify in the script a command creating an object that can be easily identified:

```
Get-Service | Out-File "C:\Users\Public\Documents\Testoutput.txt"
install-windowsfeature telnet-client
```

At the end of the deployment, after RDPing to the VM, if you find something like this, it means that you made it:



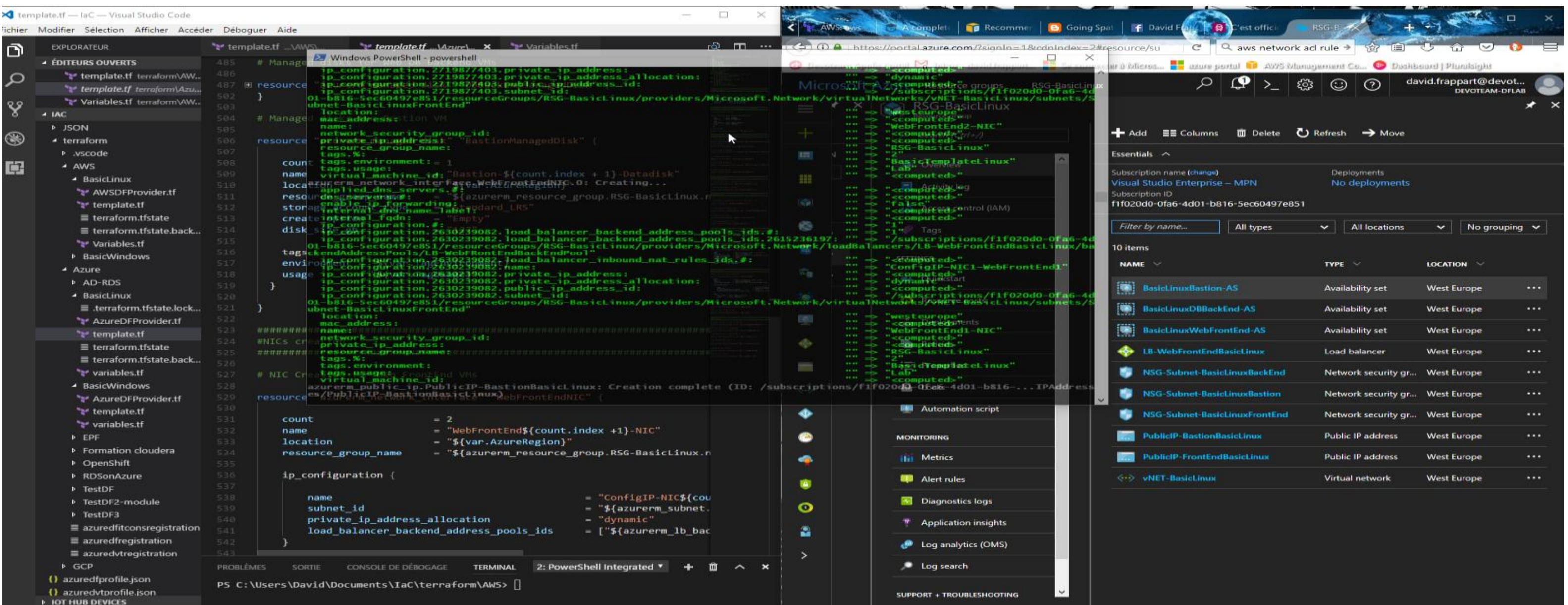
```
Status   Name               DisplayName
------   ----               -----------
Stopped  AJRouter           AllJoyn Router Service
Stopped  ALG                Application Layer Gateway Service
Running  AppHostSvc         Application Host Helper Service
Stopped  AppIDSvc           Application Identity
Stopped  Appinfo            Application Information
Stopped  AppMgmt            Application Management
Running  AppReadiness       App Readiness
Stopped  AppVClient         Microsoft App-V Client
Stopped  AppXSvc            AppX Deployment Service (AppXSVC)
Stopped  AudioEndpointBu... Windows Audio Endpoint Builder
Stopped  Audiosrv           Windows Audio
Stopped  AxInstSV           ActiveX Installer (AxInstSV)
Stopped  BDESVC             BitLocker Drive Encryption Service
Running  BFE                Base Filtering Engine
Stopped  BITS               Background Intelligent Transfer Ser...
```

4 Conclusion

In this article, we dove a little deeper in the bootstrapping process in Azure. While the custom script extension is one way to push script from an http repository, we look at how to do bootstrapping when hosting, or accessing hosted, script file on http is not possible. The custom agent, with the template functionnality from terraform is a good way to provide a generalized module for custom extension. Also the custom data can be used directly for linux supported vm and we can mix both way if needed with Windows VMs, as we demonstrated. To go deeper, we should now look at how we can use variable in the template file and thus have interpolation within the JSON string. See you soon

Another Tech blog

My thoughts and experiences on Cloud related tek



```
ip_configuration.2719877403.private_ip_address:
ip_configuration.2719877403.private_ip_address_allocation:
ip_configuration.2719877403.public_ip_address_id:
ip_configuration.2719877403.subnet_id:
01-b816-5ec60497e851/resourceGroups/RSG-BasicLinux/providers/Microsoft.Network/virtualNetworks/vNET-BasicLinux/subnets/S
ubnet-BasicLinuxFrontEnd"
  location:
  mac_address:
  name:
  network_security_group_id:
  private_ip_address:
  resource_group_name:
  tags.%:
  tags.environment:
  tags.usage:
  virtual_machine_id:
azurerm_network_interface.WebFrontEndNIC.0: Creating...
  applied_dns_servers.#:
  dns_servers.#:
  enable_ip_forwarding:
  internal_dns_name_label:
  internal_fqdn:
  ip_configuration.#:
  ip_configuration.2630239082.load_balancer_backend_address_pools_ids.#:
  ip_configuration.2630239082.load_balancer_backend_address_pools_ids.2615236197:
01-b816-5ec60497e851/resourceGroups/RSG-BasicLinux/providers/Microsoft.Network/loadBalancers/LB-WebFrontEndBasicLinux/ba
ckendAddressPools/LB-WebFRontEndBackEndPool"
  ip_configuration.2630239082.load_balancer_inbound_nat_rules_ids.#:
  ip_configuration.2630239082.name:
  ip_configuration.2630239082.private_ip_address:
  ip_configuration.2630239082.private_ip_address_allocation:
  ip_configuration.2630239082.public_ip_address_id:
  ip_configuration.2630239082.subnet_id:
01-b816-5ec60497e851/resourceGroups/RSG-BasicLinux/providers/Microsoft.Network/virtualNetworks/vNET-BasicLinux/subnets/S
ubnet-BasicLinuxFrontEnd"
  location:
  mac_address:
  name:
  network_security_group_id:
  private_ip_address:
  resource_group_name:
  tags.%:
  tags.environment:
  tags.usage:
  virtual_machine_id:
azurerm_public_ip.PublicIP-BastionBasicLinux: Creation complete (ID: /subscriptions/f1f020d0-0fa6-4d01-b816-...IPAddress
es/PublicIP-BastionBasicLinux)
```

```
    count                                  = 2
    name                                   = "WebFrontEnd${count.index +1}-NIC"
    location                               = "${var.AzureRegion}"
    resource_group_name                    = "${azurerm_resource_group.RSG-BasicLinux.n

    ip_configuration {

        name                                       = "ConfigIP-NIC${cou
        subnet_id                                  = "${azurerm_subnet.
        private_ip_address_allocation              = "dynamic"
        load_balancer_backend_address_pools_ids    = ["${azurerm_lb_bac
    }
```

```
PS C:\Users\David\Documents\IaC\terraform\AWS>
```

Subscription name (change)
Visual Studio Enterprise – MPN
Subscription ID
f1f020d0-0fa6-4d01-b816-5ec60497e851
Deployments
No deployments

10 items

NAME	TYPE	LOCATION
BasicLinuxBastion-AS	Availability set	West Europe
BasicLinuxDBBackEnd-AS	Availability set	West Europe
BasicLinuxWebFrontEnd-AS	Availability set	West Europe
LB-WebFrontEndBasicLinux	Load balancer	West Europe
NSG-Subnet-BasicLinuxBackEnd	Network security gr...	West Europe
NSG-Subnet-BasicLinuxBastion	Network security gr...	West Europe
NSG-Subnet-BasicLinuxFrontEnd	Network security gr...	West Europe
PublicIP-BastionBasicLinux	Public IP address	West Europe
PublicIP-FrontEndBasicLinux	Public IP address	West Europe
vNET-BasicLinux	Virtual network	West Europe