Configuration File Comparison

Tasio Méndez Ayerbe
tasio.mendez@mail.polimi.it
Politecnico di Milano

Introduction

This presentation introduces the differences between the configuration file of Terraform (HCL) and Cloudify (TOSCA). The whole configuration files can be found on GitHub.

https://github.com/leonardoreboucas/cloud-orchestrators-exam.git

On the previous repository, the files for deploying can be found using different providers. The architecture proposed consists of running WordPress using three virtual machines, one as a MySQL database server and the other two as Apache/PHP and WordPress application servers installed.

For the proper functioning of this environment, it was also necessary to configure the virtual interconnect resources (network, subnet, IPs, interfaces and routes) and security (firewall rules).

We will be focusing on **Azure** for the analysis without considering the variables and secrets which are inputs for both tools. On the repository some instructions are included in order to deploy and test both orchestrators.

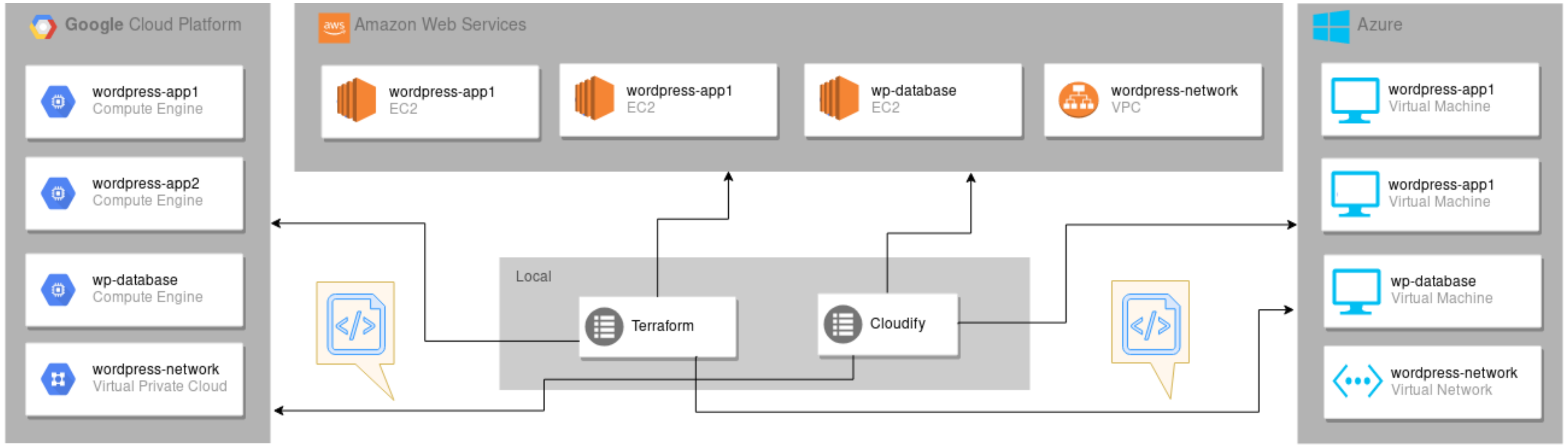
Features Comparison

This comparative table presents the relevant features of both tools in order to be able to see the comparison at a sight.

Orchestrator	Integrable with	OpenSource	Artifact	CLI	API	Graphic UI
Terraform	AWS, GCP, Azure, OpenStack, Oracle, vSphere, and more than 30	Yes	HCL JSON	Yes	Online functions	No
Cloudify	AWS, GCP, Azure, OpenStack, vCloud and vSphere	Yes	TOSCA YAML	Yes	Yes	Yes

Scenario Proposed

For this research, the following architecture was used for running Wordpress and a MySQL Server by using similar machines to mitigate the interference of differences in the provision of services.



Provider Configuration

The provider configuration is quite similar on both orchestrators. However, on Cloudify, it should be included on every node which can be done by defining a DSL Definition.

```
provider "azurerm" {
  subscription_id = "${var.azure_subscription_id}"
  tenant_id       = "${var.azure_tenant_id}"
  client_id       = "${var.azure_client_id}"
  client_secret   = "${var.azure_client_secret}"
}
```

The variable definition can be done by defining input variables or secrets on both orchestrators. However, it should be done by defining input secrets.

```
dsl_definitions:

  azure_config: &azure_config
    subscription_id: { get_secret: azure_subscription_id }
    tenant_id: { get_secret: azure_tenant_id }
    client_id: { get_secret: azure_client_id }
    client_secret: { get_secret: azure_client_secret }

# Using the previous dsl_definition on a given node
node_name:
  type: ...
  properties:
    ...
    azure_config: *azure_config
    ...
```

Resource Group

The Resource Group is the main element of the architecture as it will aggregate all the elements from Azure which are used such as the Virtual Network or Virtual Machines.

The availability set provide VM redundancy and availability. This configuration within a datacenter ensures that during either a planned or unplanned maintenance event, at least one virtual machine is available and meets the 99.95% Azure SLA.

```
resource "azurerm_resource_group" "main" {
  name     = "wordpress-${random_id.bp_suffix.hex}"
  location = "${var.region_name}"
}

resource "azurerm_availability_set" "wordpress-availability" {
  name                = "wordpress-availability-${random_id.bp_suffix.hex}"
  location            = "${azurerm_resource_group.main.location}"
  resource_group_name = "${azurerm_resource_group.main.name}"
  managed             = true
}
```

Resource Group

```
resource_group:
  type: cloudify.azure.nodes.ResourceGroup
  properties:
    name: { get_input: resource_group_name }
    location: { get_input: azure_region_name }
    azure_config: *azure_config

availability_set:
  type: cloudify.azure.nodes.compute.AvailabilitySet
  properties:
    name: {concat:['09',{get_input: resource_prefix},availabilityset,{get_input: resource_suffix}]}
    location: { get_input: azure_region_name }
    azure_config: *azure_config
    retry_after: { get_input: retry_after }
  relationships:
  - type: cloudify.azure.relationships.contained_in_resource_group
    target: resource_group
```

Virtual Network

The first configuration of the network layer should be the virtual network and the subnet which is linked to the the virtual network. Then, we would need to define the interfaces for the virtual machines as well as the public IPs in order to be able to access WordPress from a browser.

```
resource "azurerm_virtual_network" "main" {
  name                = "wordpress-network-${random_id.bp_suffix.hex}"
  address_space       = ["10.0.0.0/16"]
  location            = "${azurerm_resource_group.main.location}"
  resource_group_name = "${azurerm_resource_group.main.name}"
}

resource "azurerm_subnet" "internal" {
  name                 = "wordpress-subnet-${random_id.bp_suffix.hex}"
  resource_group_name  = "${azurerm_resource_group.main.name}"
  virtual_network_name = "${azurerm_virtual_network.main.name}"
  address_prefix       = "10.0.2.0/24"
}
```

Virtual Network

```
network:
  type: cloudify.azure.nodes.network.VirtualNetwork
  properties:
    name: { get_input: network_name }
    resource_group_name: { get_input: resource_group_name }
    azure_config: *azure_config
    location: { get_input: azure_region_name }
  relationships:
  - type: cloudify.azure.relationships.contained_in_resource_group
    target: resource_group

subnet:
  type: cloudify.azure.nodes.network.Subnet
  properties:
    name: { get_input: subnet_name }
    resource_group_name: { get_input: resource_group_name }
    azure_config: *azure_config
    location: { get_input: azure_region_name }
    resource_config:
      addressPrefix: 10.10.0.0/24
  relationships:
  - type: cloudify.azure.relationships.contained_in_virtual_network
    target: network
```

Virtual Machines

The data source on Terraform *template_file* will take a bash script template which will be in charge of installing and configuring the MySQL server or WordPress on the virtual machine. On the other side, Cloudify uses a node that will run an ansible script in charge of applying the same changes.

As we will see, Terraform uses a resource from Azure which is called Virtual Machine Extension to run the script which provide post-deployment configuration and automation. However, Cloudify will make use of the lifecycle of nodes to run the ansible script. At the end will have the same results.

The configuration of the interfaces for all the virtual machines will be quite similar on both orchestrators as well as the profile and storage configuration.

The configuration for WordPress and MySQL Virtual Machines will be the same. The difference come from the scripts that will run the installation on them.

VM Configuration

Terraform provides provisioners to configure a Virtual Machine. By default, provisioners run when the resource they are defined within is created. Creation-time provisioners are only run during creation, not during updating or any other lifecycle. They are meant as a means to perform bootstrapping of a system.

- A provisioner can **load a file** into a given Virtual Machine, as well as executing **remote executions**. By copying and executing a file into the remote component such as a bash script we can run a configuration script which installs all the dependencies.

- However, a provisioner can also be **executed locally**, which means that we can run also some other auto configuration tools such as Ansible playbooks.

```
provisioner "local-exec" {
  command = "ansible-playbook -u root ... some-ansible-playbook.yml"
}
```

VM Configuration

```
provisioner "file" {
  connection {
    host        = "${azurerm_public_ip.static-ip2.ip_address}"
    type        = "ssh"
    user        = "${var.admin_username}"
    private_key = file("${path.module}/../../common/wordpress.pem")
  }
  content     = "${data.template_file.config.rendered}"
  destination = "/tmp/config.sh"
}

provisioner "remote-exec" {
  connection {
    host        = "${azurerm_public_ip.static-ip2.ip_address}"
    type        = "ssh"
    user        = "${var.admin_username}"
    private_key = file("${path.module}/../../common/wordpress.pem")
  }
  inline = [
    "sudo chmod +x /tmp/config.sh",
    "sudo /tmp/config.sh",
  ]
}
```

VM Configuration

```
provisioner "remote-exec" {
  inline = ["sudo apt update", "sudo apt install python3 -y", "echo Done!"]
  connection {
    host        = "${azurerm_public_ip.static-ip2.ip_address}"
    type        = "ssh"
    user        = "${var.admin_username}"
    private_key = file("${path.module}/../../common/wordpress.pem")
  }
}

provisioner "local-exec" {
  command = <<EOT
      ANSIBLE_HOST_KEY_CHECKING=False \
      ansible-playbook -u ${var.admin_username} -i '${azurerm_public_ip.static-ip2.ip_address},' \
      --private-key '${path.module}/../../common/wordpress.pem' \
      --extra-vars "{ \
        "database_host": "${azurerm_public_ip.static-ip-database.ip_address}", \
        "database_name": "${var.db_name}", \
        "database_user": "${var.db_user}", \
        "database_password": "${var.db_pass}", \
      }" \
      ./playbook.yaml
   EOT
}
```

VM Configuration

Configuration in **TOSCA** can be done by Ansible by connecting a new node which will be in charge of executing the configuration playbook.

```
config-database:
  type: cloudify.nodes.Root
  interfaces:
    cloudify.interfaces.lifecycle:
      configure:
        implementation: ansible.cloudify_ansible.tasks.run
        inputs:
          site_yaml_path: wordpress/playbook_database.yaml
          sources: { get_attribute: [ SELF, sources ] }
          run_data:
            database_name: { get_input: db_name }
            database_user: { get_input: db_user }
            database_password: { get_input: db_pass }
  relationships:
    - type: cloudify.ansible.relationships.connected_to_host
      target: vm-database
```

Thanks

Tasio Méndez Ayerbe
tasio.mendez@mail.polimi.it

Politecnico di Milano

POLITECNICO
MILANO 1863

References

de Carvalho, L. R., & de Araujo, A. P. F. (2020, May). *Performance Comparison of Terraform and Cloudify as Multicloud Orchestrators*. In 2020 20th IEEE/ACM International Symposium on Cluster, Cloud and Internet Computing (CCGRID) (pp. 380-389). IEEE.

Wurster, M., Breitenbücher, U., Falkenthal, M., Krieger, C., Leymann, F., Saatkamp, K., & Soldani, J. (2020). *The essential deployment metamodel: a systematic review of deployment automation technologies*. SICS Software-Intensive Cyber-Physical Systems, 35(1), 63-75.

Terraform by HashiCorp. *Terraform Documentation*. Retrieved December, 2020, from https://www.terraform.io/docs/

Cloudify. *Cloudify Documentation*. Retrieved December, 2020, from https://docs.cloudify.co/latest/

Leonardo Rebouças de Carvalho. GitHub. Retrieved December, 2020, from https://github.com/leonardoreboucas/cloud-orchestrators-exam