Locals and outputs Lab 1.6

Use locals and functions to generate a unique value, and add a couple of outputs.

Overview

In this lab you will

- work with `terraform console`
- use a local variable and functions
- add outputs for the FQDN and public IP address

Starting point

Your files should look similar to this:

- provider.tf

```
terraform {
  required_providers {
    azurerm = {
      source  = "hashicorp/azurerm"
      version = "~>3.1"
    }
  }
}

provider "azurerm" {
  features {}

  storage_use_azuread = true
}
```

- variables.tf

```
variable "resource_group_name" {
  description = "Name for the resource group"
  type        = string
  default     = "terraform-basics"
}

variable "location" {
  description = "Azure region"
  type        = string
  default     = "West Europe"
}

```

```
variable "container_group_name" {
  description = "Name of the container group"
  type        = string
  default     = "terraform-basics"
}

variable "prefix" {
  description = "Prefix string to ensure FQDNs are globally unique"
  type        = string
}
```

- main.tf

```
resource "azurerm_resource_group" "basics" {
  name     = var.resource_group_name
  location = var.location
}

resource "azurerm_container_group" "example" {
  name                = var.container_group_name
  location            = azurerm_resource_group.basics.location
  resource_group_name = azurerm_resource_group.basics.name
  ip_address_type     = "Public"
  dns_name_label      = "${var.prefix}-${var.container_group_name}"
  os_type             = "Linux"

  container {
    name   = "inspectorgadget"
    image  = "jelledruyts/inspectorgadget:latest"
    cpu    = "0.5"
    memory = "1.0"

    ports {
      port     = 80
      protocol = "TCP"
    }
  }
}

```

- terraform.tfvars

```
location = "UK South"
prefix = "terrauser"
```

⚠ You should have specified a different value for *prefix*. You may have used a different value for *location*.

Functions

In this section we will remove the var.prefix and instead use a substring of a predictable hash. This will give us sufficient random characters to add to properties such as FQDNs and be confident that they will be globally unique.

Using a hash substring is closer in behaviour to the commonly used *uniqueString()* function in ARM templates. The resourceId of a resource group is commonly used as a seed. It contains the subscription ID, so meets the uniqueness requirement, yet will always produce the same result in your config even if you were to destroy the environment and start again, which is a good fit for idempotency.

If you wanted to have a newly generated random string each time then look at random_id, which is in the random provider available in the Terraform Registry. OK, let's work out how to find the right function and

1. Search on "terraform functions" in your browser

   You should find the Terraform Built-in Functions page.

   Feel free to browse the available functions.

2. Select the Hash and Crypto Functions

3. Select the *sha1* function

   The sha1 encryption is not the strongest from a security perspective, but we only need this to provide a hash from the given string.

The function call is `sha1("string")`.

Terraform console

We now know the function and how to call it. Test it out in the Terraform console.

1. List the objects in state
2. `terraform state list`

   Example output:

   ```
   azurerm_container_group.example

   azurerm_resource_group.basics
   ```

   The correct Terraform identifier for the resource group is `azurerm_resource_group.basics`. You will need that later when working out how to get the resource ID for the resource group.

3. Open the Terraform console
4. `terraform console`

This is an interactive console which is ideal for testing expressions and interpolation and for interrogating the state.

Use `CTRL+D` at an empty prompt to exit the console. This is the end of file (EOF) character.

5. Test the *sha1* function

6. `sha1("This is a string")`

   Expected hash:

```
"f72017485fbf6423499baf9b240daa14f5f095a1"
```

   If you use the up arrow to recall the command you can see if will always return the same hash.

7. Find the resource ID attribute for the resource group

   Remember when you looked at the documentation page for azurerm_resource_group that id was one of the attributes.

   `azurerm_resource_group.basics`

   Example response:

```
{

  "id" = "/subscriptions/2ca40be1-7e80-4f2b-92f7-
06b2123a68cc/resourceGroups/terraform-basics"

  "location" = "uksouth"

  "name" = "terraform-basics"

  "tags" = tomap({})

  "timeouts" = null /* object */

}
```

8. Drill down to the id attribute

9. `azurerm_resource_group.basics.id`

   Example response:

```
"/subscriptions/2ca40be1-7e80-4f2b-92f7-
06b2123a68cc/resourceGroups/terraform-basics"
```

   Your response will contain a different subscription ID.

10. Retest *sha1* with the resourceId

11. `sha1(azurerm_resource_group.basics.id)`

    Example output:

    ```
    "c3818179c2946e2352e5210e826239e65f5c3396"
    ```

    That is a little long. Eight characters should be sufficient to make it unique.

    Your response will be different as your resource group ID contains a different subscription ID.

12. Find a suitable function

    Mini **challenge**!

    Search for a Terraform function to take a substring.

    - What is the name of the function?
    - What are the three arguments?

13. Extract the first eight characters from the hash

    Enter the following into the `terraform console`:
    `substr(sha1(azurerm_resource_group.basics.id), 0, 8)`

    Example output:

    ```
    "c3818179"
    ```

    Your output will be unique to you. It will also be entirely predictable. We have the right expression.

14. Exit the console

    Use `CTRL+d` to leave the console prompt and go back to the shell.

The console is great for forming and testing expressions. We have settled on `substr(sha1(azurerm_resource_group.basics.id), 0, 8)` which will be used as a suffix. Let’s declare the local.

Locals

The variables we have been using so far are more accurately called input variables, i.e. `var.<variable_name>`. (Close to the *parameters* in ARM and Bicep templates.)
It is common to place more complex expression into locals so that the main resource and data source blocks are more readable. (Much like the *variables* in ARM and Bicep templates.)

We’ll define a local variable for the unique value, and call it *uniq*. Local variables are defined in a locals block.

1. Add the following block to the top of your main.tf

```
2. locals {
3.   uniq = substr(sha1(azurerm_resource_group.basics.id), 0, 8)
4. }
```

5. Modify value for dns_name_label

    Update the interpolation expression for dns_name_label in the azurerm_container_group resource:

    ```
      dns_name_label = "${var.container_group_name}-${local.uniq}"
    ```
    Note that locals are referenced as `local.<local_name>`.

6. **Delete** the prefix variable

    This is no longer required. Remove it from variables.tf and terraform.tfvars.

Outputs

There are a couple of attributes whose values are unknown until they;ve been created. One is the public IPv4 address, and the other is the fully qualified domain name now that it is suffixed with the *uniq* local variable.

The convention with Terraform is to place all outputs in a file called outputs.tf. The lab will walk you through creating one and then you’ll be challenged to do the other.

ip_address

1. Open the console
2. `terraform console`
3. Query the Azure Container Instance
4. `azurerm_container_group.example`
5. Use dot notation to get the public IPv4 address
6. `azurerm_container_group.example.ip_address`

   This should return you container instance's current public IP address.

7. Create a file called outputs.tf and add the following block

```
8.  output "ip_address" {
9.    value = azurerm_container_group.example.ip_address
10. }
```

fqdn

OK, time for another little challenge section.

💪 **Challenge**: Add the *fqdn* output.

- Add another output, called *fqdn*
- Set the value to the azurerm_container_group's fqdn attribute, prefixed with `http://`.

Terraform workflow

Run through the terraform workflow.

1. Format the files
2. `terraform fmt`
3. Validate the config
4. `terraform validate`
5. Check the plan
6. `terraform plan`

   ⚠ Note that the ACI will be deleted and recreated due to the name change.

7. Apply the changes
8. `terraform apply`

Note that the two values are now shown as outputs.

Viewing outputs

- Display all outputs
- `terraform output`
- Display a single output
- `terraform output fqdn`
- Set a shell variable to an output value

  Bash:

  ```
  ip_address=$(terraform output -raw ip_address)
  ```

  Powershell:

  ```
  $ip_address = (terraform output -raw ip_address)
  ```

- Read all objects as JSON / Powershell object

  This is slightly more advanced.

  If you are dealing with more complex arrays and objects tjen you can read the whole output into a JSON string in Bash, or an object in PowerShell.

  You can then view the whole object, or drill in using jq and object dot notation respectively.

  Bash:

  ```
  outputs=$(terraform output -json)
  jq -Mc . <<< $outputs
  jq . <<< $outputs
  jq -r .ip_address.value <<< $outputs
  ```

  Powershell:

  ```
  $outputs = (terraform output -json | ConvertFrom-Json)
  $outputs
  $outputs | ConvertTo-Json
  ($outputs).ip_address.value
  ```

Summary

You have started to use the Terraform console, and made use of locals and outputs.

You will use both of these more and more as your Terraform code becomes more ambitious. Locals are important for readability as the expressions become more complex. Outputs are useful at this level, but become far more important when you start creating your own modules.

In the next section we will manipulate state a little using imports, refresh, renames and taints.
