Introduction

Contents

Overview

Terraform allows you to control your infrastructure on a cloud service provider.

The big providers are:

- AWS
- Azure
- GCP

Infrastructure as Code

Infrastructure as Code allows us to use a high level of descriptive programming language to describe and manage infrastructure.

There is a known issue with pipelines called **environment drift**, which means that over time, an environment can end up in a unique configuration that cannot be automatically recreated. When environments become inconsistent, deployments can be affected and testing can be made invalid.

With infrastructure as code, the infrastructure configurations can be versioned and maintained, so if another environment needs to be created, you can be sure that you are using up to date configurations.

Workflows

There are a few steps that should be followed for a deployment - don't worry if you don't understand the concepts immediately, as they will make more sense as we continue through the module.

1. **Scope** - check resources that need to be created for a given project
2. **Author** - create the configuration file
3. **Initialise** - execute `terraform init` in the project directory where the configuration file lies.
   This will download any dependencies necessary for the selected cloud provider.
4. **Plan** - execute `terraform plan` in the project directory where the configuration file lies.
   This will verify the creation process and scan the configuration file for any detectable faults.
5. **Apply** - execute `terraform apply` in the project directory where the configuration file lies.
   This will create the actual resource as well as the state file which Terraform will use to check for changes in the configuration file to what is actually deployed.

Common use cases

- **Multi-Tier Applications** - It is very common to have applications with multiple tiers, each tier having different requirements and dependencies.
  With Terraform we are able to describe each tier of the application as a collection of resources so that the dependencies for each tier can be handled automatically.
- **Software Demos** - Although tools like Vagrant can be used to create environments for demos, vagrant can’t completely mimic production environments.
  Additionally, depending on how large the infrastructure for the application is, it might be challenging to run it on something like a laptop.
  Because configurations for Terraform can be distributed, demos can be run against the end user’s infrastructure with ease.
  Parameters for the Terraform configurations can also be tweaked so that the software can be demoed at any scale.
- **Disposable Environments** - It is common to have a staging environment before deploying to production, which is a smaller clone of the production environment.
  Production environments over time can become more complex and require more effort to mimic on a smaller scale.

  With Terraform, the infrastructure will be kept as code and can easily be applied to other new environments for testing and then be disposed of. Spinning up and disposing of environments easily means that costs can also be saved on environments that do not need to be operational 24/7 and only for a fraction of that time.
- **Multi-Cloud Deployments** - Terraform has the ability to configure infrastructure across more than one cloud service.
  This hybrid solution might be due to a customer wanting to take advantage of the features available on different cloud provider solutions. Another reason for multi-cloud deployments could be for extra fault tolerance.

Tutorial

Installation - Windows

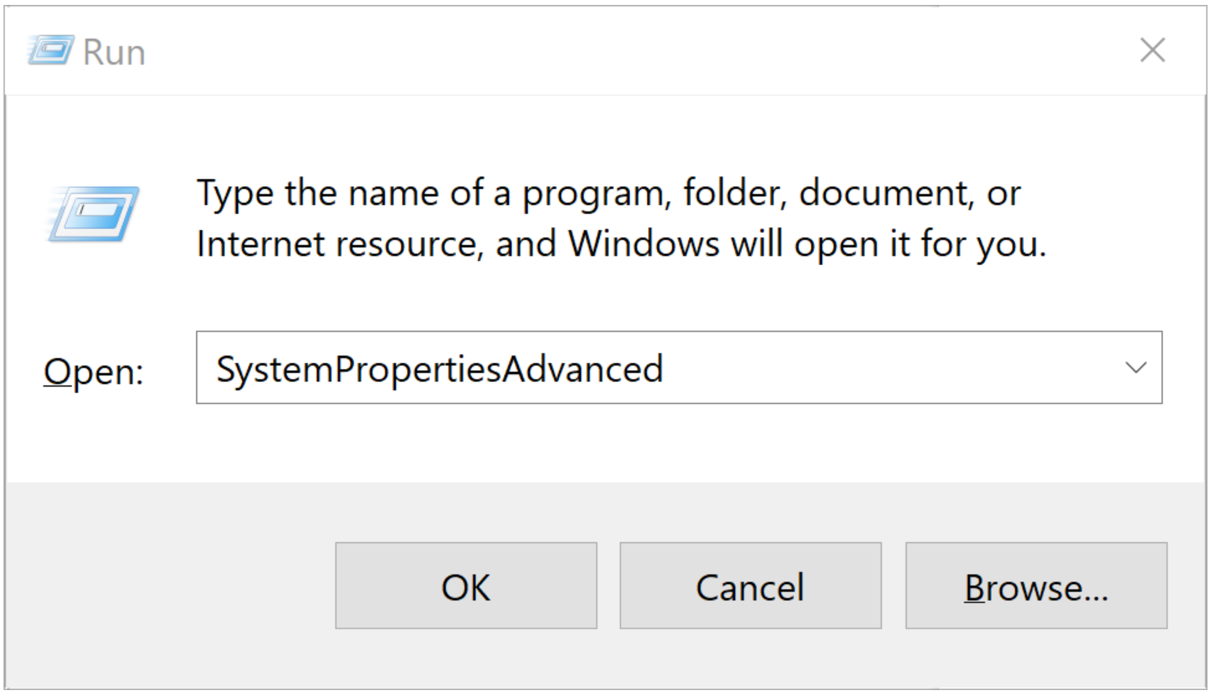
Show Tutorial

- Navigate to https://www.terraform.io/downloads.html in a web browser and download Terraform for 64-bit windows
- Extract the .zip file
- Copy the terraform.exe file from where you decided to extract it to a new folder: `C:\tools\terraform\`

Configuring the Terraform on your `PATH`

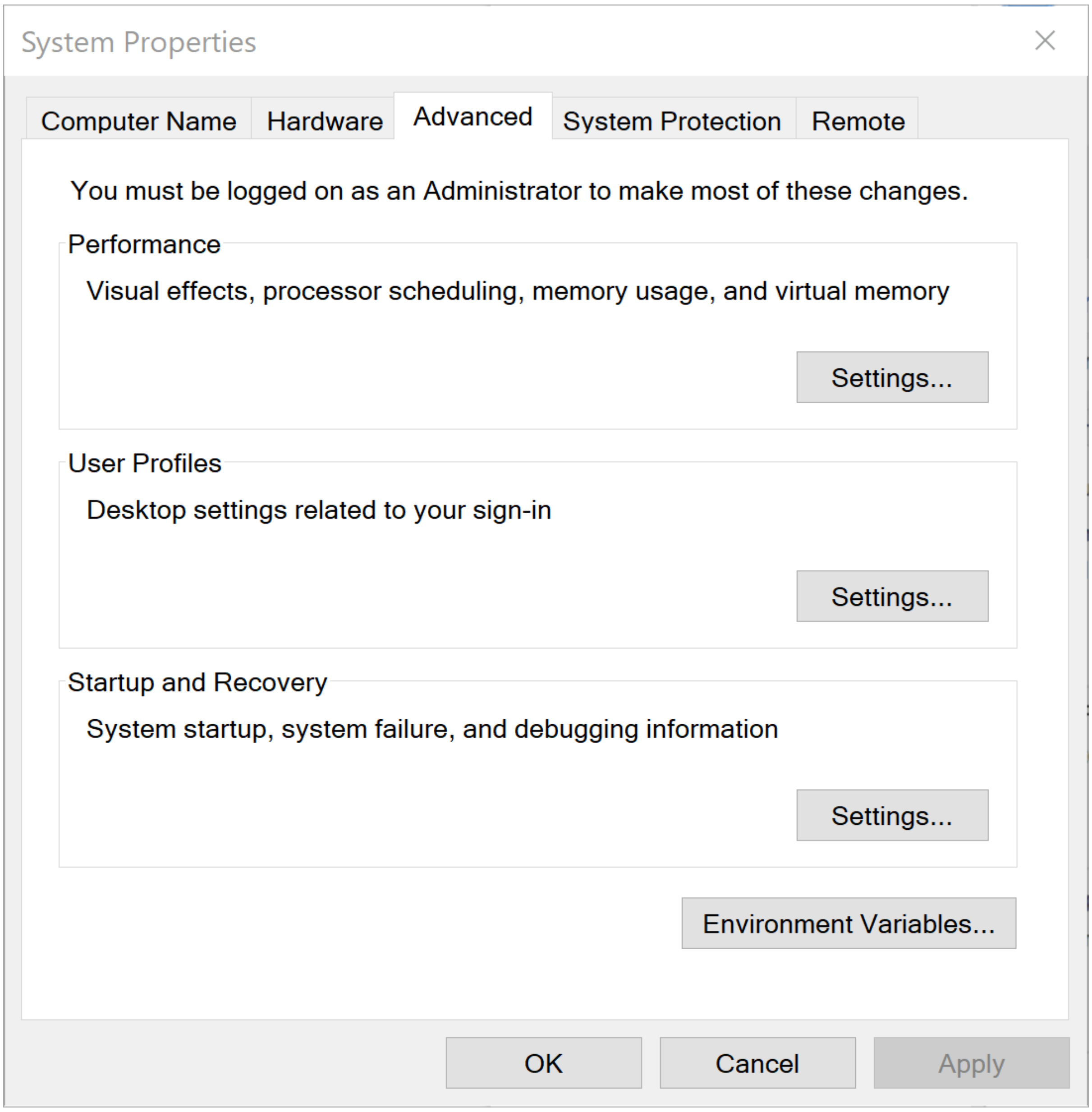
We now need to configure the `PATH` environment variable so that Terraform can be used easily on the command line.

- Press **Windows key + R** to open the **Run program**
- Type **SystemPropertiesAdvanced** and click **OK**


Run
Type the name of a program, folder, document, or Internet resource, and Windows will open it for you.
Open:
OK
Cancel
Browse...

SystemPropertiesAdvanced

- Select **Environment Variables...** button


System Properties

Computer Name | Hardware | Advanced | System Protection | Remote

You must be logged on as an Administrator to make most of these changes.

Performance

Visual effects, processor scheduling, memory usage, and virtual memory

Settings...

User Profiles

Desktop settings related to your sign-in

Settings...

Startup and Recovery

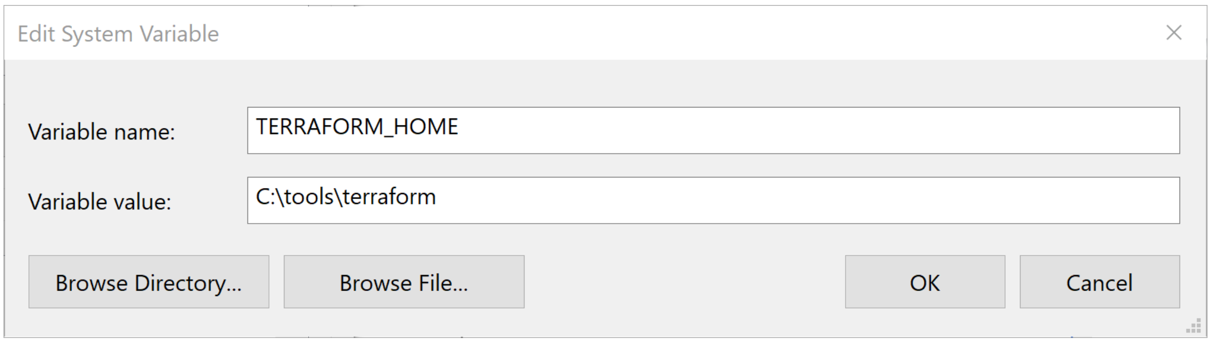
System startup, system failure, and debugging information

Settings...

Environment Variables...

OK | Cancel | Apply

- Under *User Variables for* , select **New...** and enter the Variable name: **TERRAFORM_HOME** and the Variable value: `C:\tools\terraform`, then click **OK** button

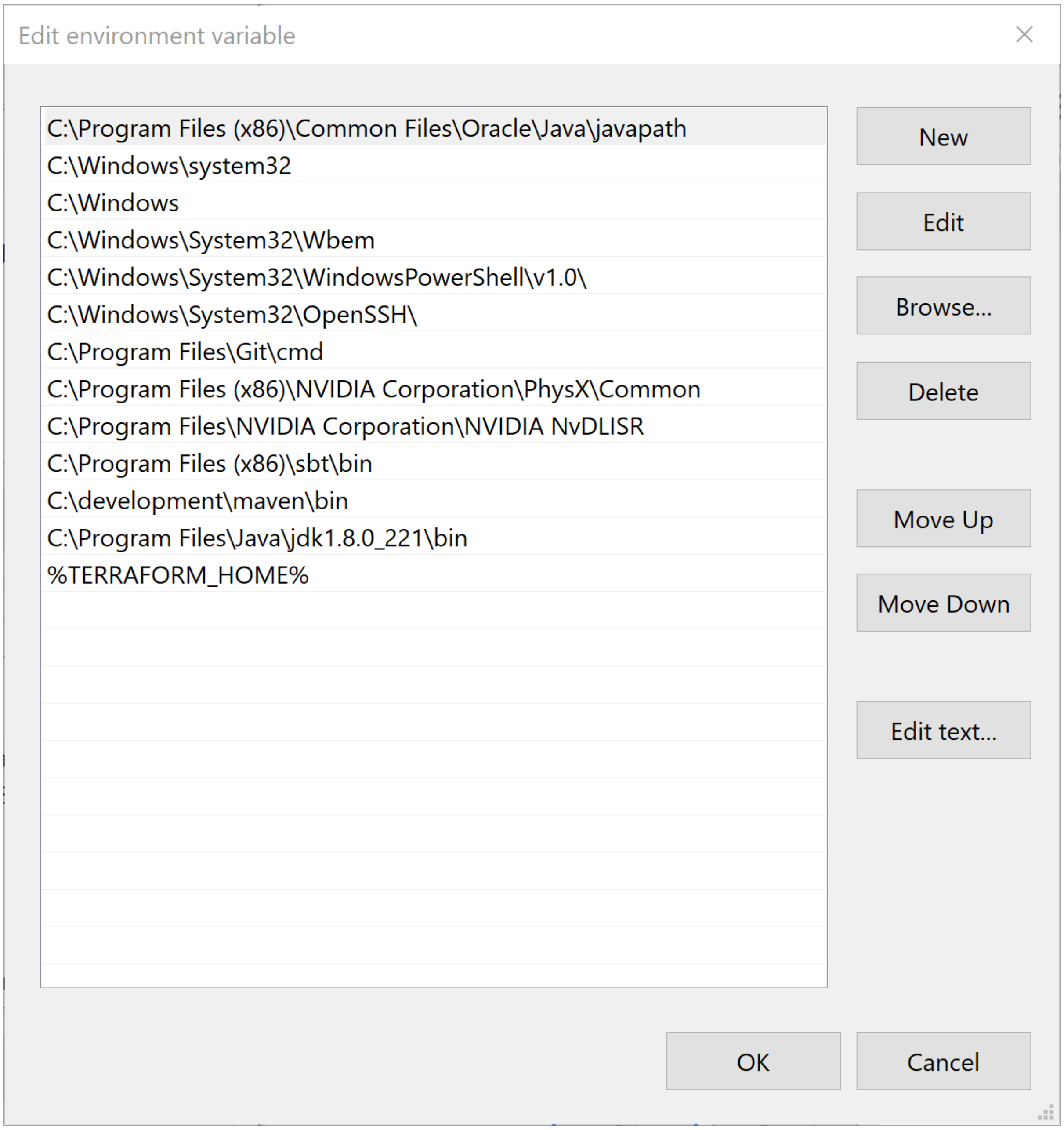

Edit System Variable

Variable name: TERRAFORM_HOME

Variable value: C:\tools\terraform

Browse Directory... | Browse File... | OK | Cancel

- Under *System Variables*, select the variable called **Path** then click **Edit...** than in the next window click **New** and enter **%TERRAFORM_HOME%**



```
C:\Program Files (x86)\Common Files\Oracle\Java\javapath
C:\Windows\system32
C:\Windows
C:\Windows\System32\Wbem
C:\Windows\System32\WindowsPowerShell\v1.0\
C:\Windows\System32\OpenSSH\
C:\Program Files\Git\cmd
C:\Program Files (x86)\NVIDIA Corporation\PhysX\Common
C:\Program Files\NVIDIA Corporation\NVIDIA NvDLISR
C:\Program Files (x86)\sbt\bin
C:\development\maven\bin
C:\Program Files\Java\jdk1.8.0_221\bin
%TERRAFORM_HOME%
```

- Click **OK** button on the *Environment Variable Windows* and close the *System Properties* window.

Verify the Installation

You can verify that you have installed Terraform correctly by running `terraform --version`.

Tasks

Azure

You will now create your first resource on Azure using Terraform.

Prerequisites

1. Have `az` CLI installed and be logged in using `az login`
2. Have `terraform` installed

Creating the directory structure

From your home directory, run the commands:

```
mkdir terraform-az-tutorial-intro && cd $_
touch main.tf
```

Your directory structure should now look like this:

```
.
└── main.tf
```

Setting up

Configure your file as following:

`main.tf`

```
terraform {
    required_providers {
        azurerm = {
            source  = "hashicorp/azurerm"
            version = "~> 2.46.0"
        }
    }
}

provider "azurerm" {
    features {}
}

resource "azurerm_resource_group" "main" {
    # creating a resource group named "my-first-tf-rg"
    name     = "my-first-tf-rg"
    location = "uksouth"
}
```

Running the configuration

First, make sure you're in the root directory for this configuration (the one with your `main.tf`).

Now, follow these steps:

1. Install the plugins we've specified: `terraform init`
2. Check what Terraform plans to do with this configuration and check it's correct: `terraform plan`
3. Apply this configuration, and build our resources: `terraform apply`

Ensure that you check the changes that this action will make to your infrastructure and then type `yes` to agree.

You should see a resource group get created.

We can check this worked correctly by running `az group list -o table` in our CLI.

Clean Up

Run `terraform destroy` to destroy all created resources.

You should get an error saying the resource can't be destroyed. In `main.tf`, change the lifecycle of the resource to

Ensure that you check the changes that this action will make to your infrastructure and type `yes` to agree.

Basic Syntax

Contents

Overview

Terraform uses its own configuration language called **Hashicorp Configuration Language** (HCL).

It's designed to describe infrastructure in a concise way.

The language is declarative, meaning it describes an intended goal rather than the steps of how to reach that goal.

Blocks

```
<type> "<label>" "<label>" {
    # Block body
    <identifier> = <expression> # Argument
}
```

Blocks are containers for arguments, which are simply key/value pairs. Blocks have a type and typically have labels that are used to reference the block.

Almost everything in HashiCorp Configuration Language (HCL) is written in blocks.

Provider

Every Terraform configuration requires you to set a **provider**. This provider will specify which plugins Terraform will download whenever you run `terraform init`.

Make sure you define this at the top of your configurations.

Example in AWS

Example in Azure

Resources

Resources are types of blocks that represent your cloud resources.

All Azure resources have two labels: the first defines the type of resource you're provisioning, and the second is a unique identifier for that specific resource.

We can then reference these blocks elsewhere within our configuration using `<resource_type>.<label>`. See examples below.

We can also reference attributes in the same way, using `<resource_type>.<label>.<identifier>`.

Example in AWS

Example in Azure

Variables

Variables are used to parameterise your configurations, making them more versatile, and easier to read and understand.

Local Variables

Local variables are defined within the file they are being used in. As the name suggests, their scope is *local* to this file.

For example:

```
locals  {
    version      = 3.2
    project_name = "big-project"
}

resource "some_resource" "example" {
    name = "${local.project_name}-resource"
}
```
As you can see, the local variable's value is set in the `locals` block and referenced using `local.<variable>`.

Input Variables

Input variables are much more versatile than local variables, as they can be referenced from any file in the same module, and can be overwritten in many different ways.

Instead of settings a variable's value out-right, we instead set its default value.

A variable block is defined like so:

```
variable "project_name" {
    default = "big-project"
}
```
We can then reference this variable with `var.<label>` elsewhere in the configuration, or we can use variable substitution to format strings with `"${var.<label>}"`.

For example:

```
resource "some_resource" "example" {
    name = "${var.project_name}-resource"
}
```
This will produce `some_resource.example` with `some_resource.example.name` equal to `"big-project-resource"`.

It is good practice to keep your variable blocks in a separate file to your main code, usually named `variables.tf`. Terraform will recognise all files within the same directory that match the `.tf` file extension.

Outputs

We use **outputs** to retrieve *attributes* created during the process of provisioning our cloud infrastructure.

For example, we often won't know the value of a VM's public IP address until it has been provisioned. We can therefore use output blocks to output this information.

Here's an example of an output block that will retrieve the value of a VM's public IP address:

Example in AWS

Example in Azure

Doing so will display this value to the console output after you perform terraform apply. You can also use these outputs in other scripts.

It is good practice to keep your variable blocks in a separate file to your main code, usually named `output.tf`.

Comments

There are three types of comments supported:

- `#` used for single-line comments
- `//` also used for single-line comments, an alternative way to `#`
- `/*` start of the multi-line comment where `*/` ends the multi-line comment

`#` is the preferred way of leaving comments in the configuration files.

The formatting tool converts the `//` into `#` comments, the reason for this is that the `//` style is not idiomatic.

Order of Configuration

The order of blocks is not significant as the language is declarative.

There is only one exception to this rule which is the `provisioner` block.

Resources will be automatically processed in the correct order based on what are the relationships between the resources.

Hence, it's up to the implementer to decide on how to structure the files and how many to have.

Formatting

Although the amount of whitespace before or after the = sign doesn't have an effect on the configuration file, it's recommended to align them.

There's no need to do this manually as you can use the `terraform fmt` command to fix formatting issues.

Using this command on this code:

```
variable "version_number" {
    type  = "string"
      default    = "0.0.1"
   description =      "version number"
}
```

Will reformat to:

```
variable "version_number" {
    type        = "string"
    default     = "0.0.1"
    description = "version number"
}
```

This may seem pointless but this aids readability, which is very important - especially when working as part of a team.

Tutorial

Please complete the task below that is most convenient for you.

Azure

You will now create a resource on Azure using some functionality from this module.

Prerequisites

1. Have `az` CLI installed and be logged in using `az login`
2. Have `terraform` installed

Creating the directory structure

From your home directory, run the commands:

```
mkdir terraform-az-tutorial-basic && cd $_
touch main.tf
```

Your directory structure should now look like this:

```
.
└── main.tf
```

Setting up

Configure your file as following:

`main.tf`

```
terraform {
    required_providers {
        azurerm = {
            source  = "hashicorp/azurerm"
            version = "~> 2.46.0"
        }
    }
}

provider "azurerm" {
    features {}
}

locals {
    # defining a couple of variables to use in this file
    project_name = "tiny-project"
    location     = "uksouth"
}

resource "azurerm_resource_group" "main" {
    # creating a resource group named "tiny-project-rg"
    name     = "${local.project_name}-rg"
    location = local.location
}

output "group_name" {
    # outputting the newly created resource group's name
    value = azurerm_resource_group.main.name
}
```

Running the configuration

First, make sure you're in the root directory for this configuration (the one with your `main.tf`).

Now, follow these steps:

1. Install the plugins we've specified: `terraform init`
2. Check what Terraform plans to do with this configuration and check it's correct: `terraform plan`
3. Apply this configuration, and build our resources: `terraform apply`

Ensure that you check the changes that this action will make to your infrastructure and then type `yes` to agree.

You should see the outputted `group_name` is `tiny-project-rg`, as we expected.

We can check this worked correctly by running `az group list -o table` in our CLI.

Clean Up

Run `terraform destroy` to destroy all created resources.

Ensure that you check the changes that this action will make to your infrastructure and type `yes` to agree.

Data types

Contents

Overview

In this module, you will learn about the various data types that are available in HCL.

Strings

Strings are a *primitive* data type used to represent text. Strings are always enclosed in double quotation marks (`"`).

Here's an example of a string value:

```
variable "location" {
    default = "eu-west-1"
}
```
Both `location` and `eu-west-1` are of the string data type.

Numbers

Numbers are a *primitive* data type used to represent numbers. Different from strings they are not encased in double quotation marks.

Here's an example of a number value:

```
variable "version" {
    default = 3.2
}
```
In this example, `version`'s default value is of the number data type.

Boolean

Boolean is the final primitive data type.

It has two possible values:

- `true`
- `false`

Here is an example of a boolean:

```
variable "active" {
    default = false
}
```

You can see that in this example the value of `false` is used as the default value for the `active` variable.

Lists

Lists are a *complex* data type which is enclosed in brackets. Lists separate items of primitive types using commas.

Here's an example of a list:

```
variable "alphabet" {
    default = ["a", "b", "c", "d", "e"]
}
```

In this example, we see that the values in the list are encased within the square brackets and separated with commas.

When retrieving a value from the list you need to use the `element` function. This function takes two arguments:

1. the list we want to get the element from.
2. the index of the element - starting at zero.

Here's an example of retrieving an element from a list:

```
element(var.alphabet, 2)
```

As the list index is zero-based, using our previously defined `alphabet`, the returned value would be `c`.

Maps

Maps are a *complex* data type which is enclosed in braces and assign a collection of keys and values pairs.

Here's an example of a map:

```
variable "images" {
    default = {
        "eu-west-1" = "ami-f976839e"
        "eu-west-2" = "ami-f976839e"
    }
}
```

In this example, we can see two key/value pairs.

When retrieving a value from the map you need to use the `lookup` function. This function takes three arguments:

1. the list we want to get the element from.
2. the key associated with the value.
3. the default value to be used if there will not be a value associated with the key provided.

Here's example of retrieving a value from a map:

```
lookup(var.images, "eu-west-1", "not found")
```

Using our previously defined `images`, the returned value would be `ami-f976839e`.

Exercises

Let's test your knowledge!

1. How many data types are there?

   Show Solution

   There are `5` data types.

2. What are the primitive data types?

   Show Solution

   - `string`
   - `number`
   - `boolean`

3. What are the complex data types?

   Show Solution

   - `list`
   - `map`

Variables

Contents

Overview

Variables are a key part of the HCL syntax. We will now take a deeper look at them in order to understand them better.

This module will look solely at input variables and does not address local variables.

Defaults

The `default` key for a variable specifies what value to use if no other value is assigned.

In this example, you can see that we're providing a default value for the variable.

```
variable "ami" {
    type    = "string"
    default = "ami-f976839e"
}
```

The `default` value is also used by Terraform to infer the type of the variable. When no default is given and no type is specified, the default type of `string` is used.

Providing a value

When no `default` is provided, you *must* specify a value. If you specify a value when a `default` is provided, the specified value is used to override the `default`.

Interactive input

If no `default` is set, you may input the value in the console upon running `terraform plan` or `terraform apply`.
This may work for small exercises but is inefficient to re-enter the values every time you issue a plan or apply. It is also not applicable for automation.

Using the `-var` option

You specify the variable name followed by an equal sign and the value you want to use for the variable.
You can use many `-var` options as required, but the commands can get very long if you have more than a few variables.

Example: `terraform plan -var version=2.2 -var ami="ami-f976839e" -var location="eu-west-1"`.

Using variable files

Variable files can be included using the `-var-file` option, or they can be automatically included if they are in your current working directory and are either named `terraform.tfvars` or have a file extension of `.auto.tfvars`.

You can use multiple variable files.

Variables defined using the `-var-file` option override variables defined using automatic variable files.

Example:

`version_22.tfvars` contents:

```
version  = 2.2
ami      = "ami-f976839e"
location = "eu-west-1"
```

Command: `terraform apply -var-file version_22.tfvars`

You'll notice the syntax is the same as regular HCL syntax, but we do not need to create any blocks here.

Using environment variables

When setting Terraform variables using environment variables the name of the environment variable must begin with `TF_VAR_` and be followed by the name of the Terraform variable.

Example: `export TF_VAR_ami="ami-f976839e" && terraform apply`

Variable Precedence

When a variable is defined multiple times, the value of the variable is usually set to the highest precedent definition.

The order of precedence from highest to lowest is as follows:

1. Specified options for `-var` and `-var-files`
2. Automated `.tfvars` files
3. Environment variables
4. Default values

When variables are defined within the same precedence level, the last value is used. For example, if you define a variable in a `-var-file` and a `-var` option, the one that you specify last on the command-line will take precedence.

Note: `map` typed variables are not set to the highest precedent definition. Instead the values of the map variable definitions are merged.

Tutorial

Azure

Show Tutorial

You will now create a resource on Azure using some functionality from this module.

Prerequisites

1. Have `az` CLI installed and be logged in using `az login`
2. Have `terraform` installed

Creating the directory structure

From your home directory, run the commands:

```
mkdir terraform-az-tutorial-variables && cd $_
touch {main,variables,output}.tf
```

Your directory structure should now look like this:

```
.
├── main.tf
├── output.tf
└── variables.tf
```

Setting up

Configure your files as following:

`main.tf`

```
terraform {
    required_providers {
        azurerm = {
            source  = "hashicorp/azurerm"
            version = "~> 2.46.0"
        }
    }
}

provider "azurerm" {
    features {}
}

resource "azurerm_resource_group" "main" {
    name     = "${var.project_name}-rg"
    location = var.location
}
```

`variables.tf`

```
variable "project_name" {
    default = "default"
}

variable "location" {
    default = "uksouth"
}
```

`output.tf`

```
output "group_name" {
    value = azurerm_resource_group.main.name
}
```

Running the configuration

First, make sure you're in the root directory for this configuration (the one with your `main.tf`).

Now, follow these steps:

1. Install the plugins we've specified: `terraform init`
2. Check what Terraform plans to do with this configuration and check it's correct: `terraform plan`

3. Apply this configuration, and build our resources: `terraform apply`

Ensure that you check the changes that this action will make to your infrastructure and then type `yes` to agree.

You should see the outputted `group_name` is currently `default-rg`.

Important - clean up before running the next step with

```
terraform destroy
```

Using environment variables

Run the commands:

```
export TF_VAR_project_name="small-project"
terraform apply
```

You should see the outputted `group_name` is now `small-project-rg`. The default was overridden.

Important - clean up before running the next step with

```
terraform destroy
```

Using -var-file

Create a file `big-project.tfvars` and configure as shown below.

`big-project.tfvars`:

```
project_name = "big-project"
```

Now we can use this file by passing it to the `-var-file` option as follows:

```
terraform apply -var-file big-project.tfvars
```

You should see the outputted `group_name` is now `big-project-rg`. Both the default and environment variables were overridden.

Important - clean up before running the next step as follows:

```
 terraform destroy -var-file big-project.tfvars
```

Using .auto.tfvars files

Rename your `big-project.tfvars` file to `big-project.auto.tfvars`

Now just run

```
terraform apply
```

You should see the outputted `group_name` is still `big-project-rg`, and we didn't even need to specify the file in the CLI!

Important - clean up before running the next step with

```
terraform destroy
```

Using -var

Finally, run the following commands:

```
terraform apply -var project_name=final
```

You should see the outputted `group_name` is now `final-rg`. The default, environment variable, and automatic tfvars file were all overridden.

Important - clean up with

```
terraform destroy -var project_name=final
```

Clean Up

See each step individually for clean-up instructions for the individual step.

Use `unset TF_VAR_project_name` to remove the previously exported environment variable.

Modules

Contents

Overview

A **module** is a collection of configuration files in a directory.

Typically, a module will consist of the following files:

- `main.tf` – contains the resources
- `variables.tf` – contains the variables used in the module
- `output.tf` – contains the outputs produced by the module

Terraform recognises configuration files as any file with a `.tf` extension.

You could simply define a single directory with a `main.tf`, `variables.tf` and `output.tf` with all of your cloud infrastructure configured, but it's common to make use of a file structure to organise your configurations.

Here's an example of a file structure that makes use of multiple modules to organise its configurations:

```
.
├── main.tf
├── output.tf
├── variables.tf
├── module-1
│   ├── main.tf
│   ├── output.tf
│   └── variables.tf
└── module-2
    ├── main.tf
    ├── output.tf
    └── variables.tf
```

In this structure, the configurations in `module-1` and `module-2` are referenced in the root configuration's `main.tf` file using module blocks:

`main.tf`:

```
...

module "first_module" {
    source          = "./module-1"
    resource_name   = some_resource.example.name

}
```

The `source` value in the module blocks reference the directory containing each module's configuration files.

The other values defined beneath source are variables that can be referenced in the modules' configuration. However, they also need to be defined in the module's `variables.tf` file as empty blocks, such as:

`module-1/variables.tf`:

```
variable "resource_name" {
}
```

It is often important to use the outputs from our modules (as defined in its `output.tf` file). We can reference a module's outputs using `module.<module_name>.<output>` from our root `main.tf` file. For example:

`module-1/output.tf`:

```
output "some_output" {
    value = ...
}
```

`main.tf`:

```
module "second_module" {
    source          = "./module-2"
    first_output    = module.first_module.some_output    # referencing the output
to ./module-1
}
```

Tutorial

Azure

Show Tutorial

You will now create a resource on Azure using some functionality from this module.

Prerequisites

1. Have `az` CLI installed and be logged in using `az login`
2. Have `terraform` installed

Creating the directory structure

From your home directory, run the commands:

```
mkdir terraform-az-tutorial-modules && cd $_
mkdir vm vnet && touch {.,vm,vnet}/{main,variables,output}.tf
```

Your directory structure should now look like this:

```
.
├── main.tf
├── output.tf
├── variables.tf
├── vm
|   ├── main.tf
|   ├── output.tf
|   └── variables.tf
└── vnet
    ├── main.tf
    ├── output.tf
    └── variables.tf
```

Setting up the vm module

Configure your files as following:

`vm/main.tf`

```
resource "azurerm_linux_virtual_machine" "main" {
    name                  = "${var.project_name}-vm"
    resource_group_name   = var.group_name
    location              = var.location
    size                  = var.vm_size
    network_interface_ids = var.interface_ids

    admin_username = "adminuser"
    admin_password = "LetMeIn!"

    disable_password_authentication = false

    os_disk {
        caching              = "ReadWrite"
        storage_account_type = var.storage_size
    }

    source_image_reference {
        publisher = "Canonical"
        offer     = "UbuntuServer"
        sku       = "18.04-LTS"
        version   = "latest"
    }
}
```

`vm/variables.tf`

```
variable "project_name" {}
variable "group_name" {}
variable "location" {}
variable "interface_ids" {}

variable "vm_size" {
    default = "Standard_B1ms"
}

variable "storage_size" {
    default = "Standard_LRS"
}
```

vm/output.tf

```
output "private_ip" {
    value = azurerm_linux_virtual_machine.main.private_ip_address
}
```

Setting up the vnet module

Configure your files as following:

vnet/main.tf

```
resource "azurerm_virtual_network" "main" {
    name                = "${var.project_name}-vnet"
    address_space       = ["10.0.0.0/16"]
    location            = var.location
    resource_group_name = var.group_name
}

resource "azurerm_subnet" "main" {
    name                 = "internal"
    resource_group_name  = var.group_name
    virtual_network_name = azurerm_virtual_network.main.name
    address_prefixes     = ["10.0.2.0/24"]
}

resource "azurerm_network_interface" "main" {
    name                = "${var.project_name}-nic"
    location            = var.location
    resource_group_name = var.group_name

    ip_configuration {
        name                          = "internal"
        subnet_id                     = azurerm_subnet.main.id
        private_ip_address_allocation = "Dynamic"
    }
}
```

vnet/variables.tf

```
variable "project_name" {}
variable "group_name" {}
variable "location" {}
```

vnet/output.tf

```
output "interface_id" {
    value = azurerm_network_interface.main.id
}
```

Setting up the root module

Configure your files as following:

`main.tf`

```
terraform {
    required_providers {
        azurerm = {
            source  = "hashicorp/azurerm"
            version = "~> 2.46.0"
        }
    }
}

provider "azurerm" {
    features {}
}

resource "azurerm_resource_group" "main" {
    name     = "${var.project_name}-rg"
    location = var.location
}

module "vnet" {
    source       = "./vnet"
    project_name = var.project_name
    group_name   = azurerm_resource_group.main.name
    location     = var.location
}

module "vm" {
    source        = "./vm"
    project_name  = var.project_name
    group_name    = azurerm_resource_group.main.name
    location      = var.location
    interface_ids = [module.vnet.interface_id]
}
```

`variables.tf`

```
variable "project_name" {
    default = "example-project"
}

variable "location" {
    default = "uksouth"
}
```

`output.tf`

```
output "vm_private_ip" {
    value = module.vm.private_ip
}
```

Running the configuration

Now we just need to execute our configuration files.

First, make sure you're in the root directory for this configuration (the one with your root `main.tf`).

Now, follow these steps:

1. Install the plugins we've specified: `terraform init`
2. Check what Terraform plans to do with this configuration and check it's correct: `terraform plan`
3. Apply this configuration, and build our resources: `terraform apply`

Ensure that you check the changes that this action will make to your infrastructure and then type `yes` to agree.

Clean Up

Run `terraform destroy` to destroy all created resources.

Ensure that you check the changes that this action will make to your infrastructure and type `yes` to agree.

Intermediate Syntax

Contents

Overview

In this module, we will look at some more intermediate syntax. Sometimes the basic syntax just isn't enough for some more advanced infrastructures.

Meta Arguments

There are several meta-arguments able to be used. Each of these meta-arguments can be used in *any* resource or module block, regardless of the plugins being used.

To list a few:

- `depends_on`
- `count`
- `for_each`
- `provider`
- `lifecycle`

depends_on

`depends_on` is used to handle hidden resource dependencies when Terraform isn't able to automatically infer them.

depends_on must be a list of references to other resources.

count

By default, Terraform configures each resource or module once (per block). We may use the count argument to override this default.

count accepts a number type as its argument - representing the number of resources that should be created.

To reference one of these created resources, we would use <resource_type>.<label>[<index>].

Example:

```
resource "azurerm_resource_group" "example" {
    count = 2 # create two similar resource groups

    name     = "rg-${count.index}"
    location = "West Europe"
}
```

As you can see, a new object count is created with an attribute count.index. This index is zero-based.

for_each

If your resource instances are identical, or close to being identical, count is appropriate to use. If some resource arguments need more distinct values, then it may be more appropriate to use for_each.

for_each accepts a map as its argument.

Example in Azure

When used, a new object each, with attributes each.key and each.value, will allow access to the key and value of each element of the for_each expression.

Each infrastructure object is distinct - meaning every resource is *created*, *updated*, and *deleted* separately when the configuration is applied.

provider

Terraform allows having multiple providers defined in a configuration file, although there can only be one default provider and others will have to use aliases.

One of the potential use cases for having multiple providers could be to manage resources in different regions or use different cloud providers altogether.

The `provider` meta argument within the resource block overrides the usage of the default provider.

Example

In this example there are two providers defined, the first provider going from top to bottom will be the *default* one.

The second provider has an alias, this will be used to make a reference that this provider should be used when interacting with the resource in the resource block.

```
terraform {
    required_providers {
        azurerm = {
            source  = "hashicorp/azurerm"
            version = "~> 2.46.0"
        }
    }
}

provider "azurerm" {
    environment = "public"
}

provider "azurerm" {
    environment = "german"
    alias  = "azure-german"
}

resource "azurerm_resource_group" "example-german" {
    provider = "azurerm.azure-german"

    name     = "rgroup1"
    location = "West Europe"
}
```

lifecycle

Additional details about the resource's life-cycle can be provided in the `lifecycle` meta argument.

`lifecycle` additional meta-arguments are:

- `create_before_destroy`
- `prevent_destroy`
- `ignore_changes`

create_before_destroy

`create_before_destroy` requires a `boolean` value to be set.

Terraforms default behaviour is to destroy the resource if the requested change cannot be applied due to some limitations.

After destroying the resource a new one will be created in its place with the applied change and it will replace the initial resource.

`create_before_destroy` allows for changing this behaviour.

When `create_before_destroy` is set, a new resource will be created first with the applied change, and only then will the previous resource will be destroyed.

Example

```
resource "azurerm_resource_group" "example" {
    name     = "rgroup1"
    location = "West Europe"

    lifecycle {
        create_before_destroy = true
    }
}
```

prevent_destroy

`prevent_destroy` requires a `boolean` value to be set.

When this meta-argument is set to `true` any attempt to destroy the resource will be rejected with an error message, as long as this meta-argument is still present in the configuration file for the resource.

One of the use cases could be to prevent precious resources like a database from being destroyed.

The only downside is that by having this enabled could be harder to make changes to the resource, additionally calling `terraform destroy` once the resource was created wouldn't actually destroy this resource.

Example

```
resource "azurerm_resource_group" "example" {
    name     = "rgroup1"
    location = "West Europe"

    lifecycle {
        prevent_destroy = true
    }
}
```

ignore_changes

`ignore_changes` takes a list of attribute names.

Terraforms default behaviour is to check whether there are new changes to the current configuration files comparing it to the previous version.

There may be some cases outside of Terraform where a resource will be modified, then Terraform would attempt to fix that by setting the values back to the ones that are defined in the configuration file.

You might not always want this sort of behaviour and the `ignore_changes` allows you to ignore changes happening to some attributes.

In these examples, `tags` is passed in the list.

These examples would make any changes happening outside of Terraform to the `tags` attribute be ignored by Terraform.

Example

```
resource "azurerm_resource_group" "example" {
    name     = "rgroup1"
    location = "West Europe"

    tags = {
        Environment = "development"
    }

    lifecycle {
        ignore_changes = [tags]
    }
}
```

Operation timeouts

Certain resource types allow to define how long they can take for operations to take place, this is defined within the `resource` block.

For example `aws_instance`, allows `timeouts` for *create* (default 10min), *update* (default 10min), and *delete* (default 20min).

In these examples, the allowed time it takes for the creation of the resource is overridden to five minutes and the destruction of it is overridden to two hours.

Example

```
resource "azurerm_resource_group" "example" {
    name     = "rgroup1"
    location = "West Europe"

    timeouts {
        create = "5m"
        delete = "2h"
    }
}
```

Tutorial

You will now create a resource on Azure using some functionality from this module.

Prerequisites

1. Have `az` CLI installed and be logged in using `az login`
2. Have `terraform` installed

Creating the directory structure

From your home directory, run the commands:

```
mkdir terraform-az-tutorial-intermediate && cd $_
touch main.tf
```

Your directory structure should now look like this:

```
.
└── main.tf
```

Setting up

Choose either of the following for the configuration:

count

for_each

Running the configuration

First, make sure you're in the root directory for this configuration (the one with your `main.tf`).

Now, follow these steps:

1. Install the plugins we've specified: `terraform init`

2. Check what Terraform plans to do with this configuration and check it's correct: `terraform plan`
3. Apply this configuration, and build our resources: `terraform apply`

Ensure that you check the changes that this action will make to your infrastructure and then type `yes` to agree.

You should see two resource groups get created.

We can check this worked correctly by running `az group list -o table` in our CLI.

Clean Up

Run `terraform destroy` to destroy all created resources.

You should get an error saying the resource can't be destroyed. In `main.tf`, change the lifecycle of the resource to

```
lifecycle {
    prevent_destroy = false
}
```

Run `terraform destroy` again.

Ensure that you check the changes that this action will make to your infrastructure and type `yes` to agree.

Branch Model

Contents

Overview

In this module, we're going to look at reusing modules to support multiple environments in Terraform.

A branch model would look something like the following:

```
.
├── environments
|   ├── production
|   |   ├── production.tf
|   |   └── production.tfvars
|   └── staging
|       ├── staging.tf
|       └── staging.tfvars
└── modules
    └── infrastructure
        ├── main.tf
        ├── output.tf
        ├── variables.tf
        └── module-1
            ├── main.tf
            ├── output.tf
            └── variables.tf
      ...
```

You should already be familiar with the structure of the `infrastructure` directory from the **Modules** topic.

The `environments` directory contains the differences between our environments. We use `.tfvars` files to configure different resources based on the variables set for each environment.

The `production.tf` and `staging.tf` configurations will generally look something like:

```
variable "environment" {}
variable "instance_count" {}

terraform {
    required_providers {
        aws = {
            source  = "hashicorp/aws"
            version = "~> 3.0"
        }
    }
}

provider "aws" {
    region = "us-east-1"
}

module "infrastructure" {
    source         = "../../modules/infrastructure"
    environment    = var.environment
    instance_count = var.instance_count
}
```

We are simply setting up empty input variables which we will override with our `.tfvars` files, and importing the modules from our `infrastructure` directory.

We may vary our `instance_count` depending on if we're using *production* or *staging*, which allows us to have a cheaper *staging* environment.

To run the *staging* configuration, we simply use

```
cd environments/staging
terraform init
terraform plan -var-file staging.tfvars -out=plan # using a plan file, not
necessary
terraform apply plan
```

To run the *production* configuration, simply replace `staging` with `production` everywhere in the above script.

Tutorial

Azure

You will now create a resource on Azure using some functionality from this module.

Prerequisites

1. Have `az` CLI installed and be logged in using `az login`
2. Have `terraform` installed

Creating the directory structure

From your home directory, run the command:

```
mkdir terraform-az-tutorial-branch-model && cd $_
git clone https://gitlab.com/qacdevops/terraform-az-tutorial-branch-model.git --
branch no-env .
```

Your directory structure should now look like this:

```
.
├── README.md
└── modules
    └── infrastructure
        ├── main.tf
        ├── output.tf
        ├── variables.tf
        ├── vm
        │   ├── main.tf
        │   ├── output.tf
        │   └── variables.tf
        └── vnet
            ├── main.tf
            ├── output.tf
            └── variables.tf
```

Setting up an environment

We have our modules configured and ready to use, so now we just need to create an environment.

From the repo's main directory, run the command:

```
mkdir -p environments/tutorial && cd $_
touch tutorial.tf{,vars}
```

You should now be in the `environments/tutorial` directory, and its structure should now look like this:

```
.
├── tutorial.tf
└── tutorial.tfvars
```

Configure your files as following:

`tutorial.tf`

```
variable "project_name" {}
variable "location" {}
variable "vm_count" {}
variable "vm_size" {}
variable "storage_size" {}

terraform {
    required_providers {
        azureri = {
            source  = "hashicorp/azurerm"
            version = "~> 2.46.0"
        }
    }
}
```

```
provider "azurerm" {
    features {}
}

module "infrastructure" {
    source       = "../../modules/infrastructure"
    project_name = var.project_name
    location     = var.location
    vm_count     = var.vm_count
    vm_size      = var.vm_size
    storage_size = var.storage_size
}
```

`tutorial.tfvars`

```
project_name = "tutorial"
location     = "uksouth"
vm_count     = 2
vm_size      = "Standard_B1ms"
storage_size = "Standard_LRS"
```

Feel free to play around with the values in the `tutorial.tfvars` file.

Running the configuration

Now we just need to execute our configuration files.

First, make sure you're in the tutorial environment directory for this configuration (the one with your `tutorial.tf`).

Now, follow these steps:

1. Install the plugins we've specified: `terraform init`
2. Check what Terraform plans to do with this configuration, check its correct, and output this to a `plan` file: `terraform plan -var-file tutorial.tfvars -out=plan`
3. Apply this configuration, and build our resources: `terraform apply plan`

Go ahead and make another environment with some differing values in its `.tfvars` file and see the effects take place.

Clean Up

Run `terraform destroy -var-file tutorial.tfvars` to destroy all created resources.

Ensure that you check the changes that this action will make to your infrastructure and type `yes` to agree.

Exercises

Pick an infrastructure that you created during your specialist week and try to re-create it using Terraform.

Some Ideas:

- Load Balancer
- Virtual Machine Scale Set (VMSS)
- Function App

Create support for at least 2 environments in your configuration.