Terraform Basics

In this Module

- Getting started with terraform
  - Structure of a terraform application
  - Terraform providers and configuration
  - Terraform workflow: init, validate, plan, apply and destroy
  - Basic Hashicorp Configuration Language (HCL) syntax
    - Resources, data sources, variables and outputs
  - Introduction to terraform state
  - Working with resource arguments and attributes
  - Querying AWS data sources

Structure of a Terraform Application

- A terraform application is made up of modules
  - A module is a directory that contains terraform source files
  - Any text file with a *.tf* extension is a terraform source file
- The main module we run the *terraform* utility from is called the **root** module
  - Every terraform application has a root module
  - The terraform state file is by default in the root module directory
  - Additional modules are optional (covered later in the course)

Terraform Source Files

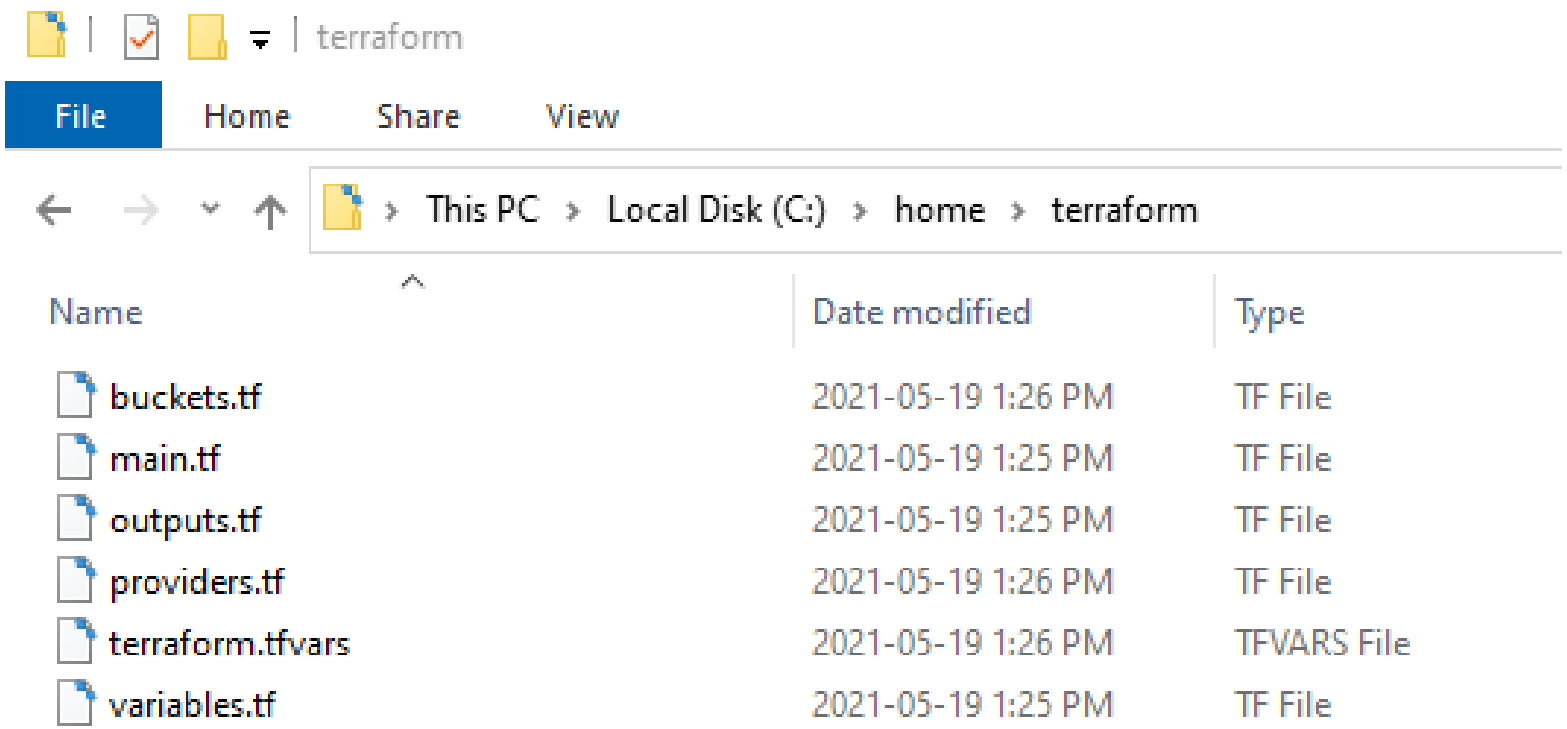
- Terraform merges the contents of all the *.tf files in a directory before doing anything
  - This means that you can name your *.tf files whatever you want
  - You can have as many *.tf files as you want in the module
- There can be an optional file *terraform.tfvars*
  - This sets the values of variables when terraform executes
  - This file must be named *terraform.tfvars*
- Terraform ignores all other files in the module that do not have *.tf extension

Canonical File Names

- The terraform community has a file naming convention that is generally adhered to
  - This makes reading terraform source code easier for other developers
- The file are:
  - *variables.tf* : contains variable definitions
  - *outputs.tf* : contains the return value (output) definitions
  - *providers.tf* : contains the provider, versioning and backend configurations
  - *main.tf* : contains the core code - resource definitions,etc.
- If the *main.tf* starts to become too difficult to read, it is often broken down into subfiles
  - For example, all S3 bucket code might be in *buckets.tf*

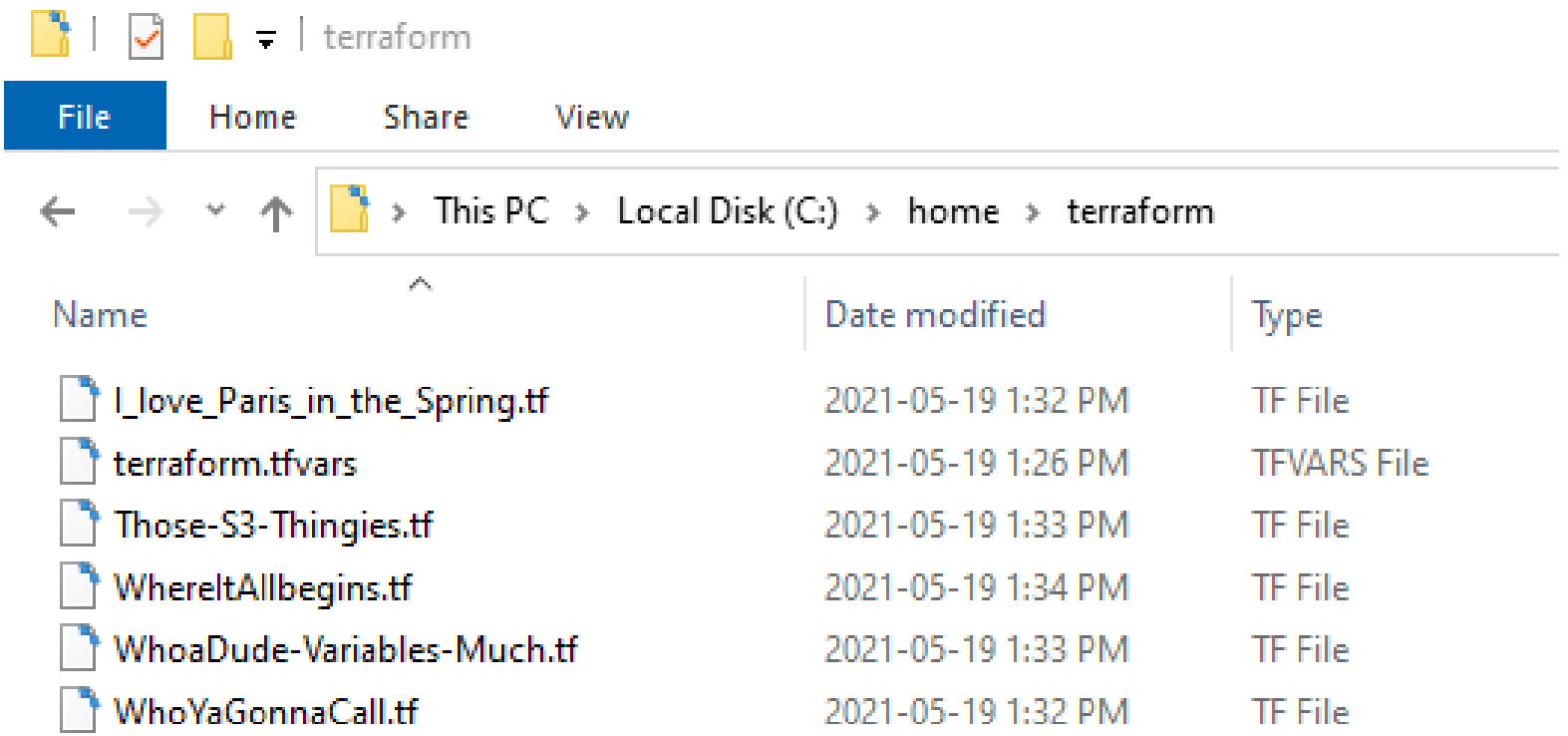
Canonical Module

- A typical terraform module looks like the screenshot below
- It is considered a professional best practice to use this structure for all terraform work
  - Additional files, like *buckets.tf* are added when they improve the readability of the code



Non-Canonical Modules

- The screenshot below shows a non-canonical module structure
  - This will still work, terraform does not care what we name the files
- Note: we cannot rename *terraform.tfvars* or terraform will ignore it



The Five Basic Terraform Constructs

- Configuration Directives: these include *provider* and *terraform*
- *resource* : specifies an AWS resource managed by terraform
- *data* : specifics a resource in the AWS environment that we want to query
- *variable* : defines an input to a terraform module
- *output* : defines a return value or output from the module

The *providers.tf* File

providers.tf > ...

```
1
2   # Example 02-01 - Configuration
3
4   terraform {
5       required_providers {
6           aws = {
7               source = "hashicorp/aws"
8               version = ">= 3.0"
9           }
10      }
11      # Required version of terraform
12      required_version = ">0.14"
13  }
14
15  provider aws {
16      region = "us-east-2"
17      profile = "dev"
18  }
```

The *terraform* Directive

- Specifies the plugins needed to communicate with the cloud vendor(s)
  - Defines the location of the plugins and versions
  - Defines location of the terraform backend (covered later)
- This directive can be omitted
  - Then defaults values will be used
  - Some defaults are inferred from the *provider* directive

The *provider* Directive

- The provider directive contains configuration information specific to a provider
  - AWS needs different configuration information (like a region) than does Azure or Google Cloud
- There can be more than one provider
  - Different providers are identified by aliases
  - The provider without an alias is the default provider
  - Multiple providers are demonstrated later

The *main.tf* File

- Two resources are defined in the file that are going to be managed by terraform
  - An EC2 instance and an S3 bucket
- The default VPC, not managed by terraform, is identified as a data source

main.tf > ...

```
1
2   # Example 02-01
3
4   resource "aws_instance" "myVM" {
5       ami = "ami-077e31c4939f6a2f3"
6       instance_type = "t2.micro"
7       tags = {
8           Name = "Example-01"
9       }
10  }
11
12  resource "aws_s3_bucket" "myBucket" {
13      bucket = "terraform-example-02-01"
14  }
15
16  data "aws_vpc" "default_VPC" {
17      default = true
18  }
19
```

The *resource* Directive Arguments

- Always start with the keyword *resource*
- Followed by a string ("aws_instance") which identifies the type of resource
- Followed by a string ("myVM") which is how the resource is referred to in the terraform code
- Followed by a list of arguments used to create the resource
  - Some arguments are mandatory, like the *ami* and *instance_type* for an EC2 instance
  - Some arguments are mandatory but have defaults, like *versioning* on S3 buckets
  - Some arguments are optional, like defined tags
- For each resource, there is a documentation page describing all the attributes associated with a specific resource
  - *https://registry.terraform.io/providers/hashicorp/aws/latest/docs/resources/instance*
  - The documentation also provides examples of how to define the resource

The *resource* Directive Attributes

- Some properties of a resource are assigned by AWS
  - These are referred to as *attributes*
  - Like public_ip of an EC2 instance or the arn of a bucket for example
  - These are defined by AWS but can be accessed by us after the resource is created
- Syntax for accessing the public IP of *myVM* for example:
  - *aws_instance.myVM.public_ip*
- Once created, all the arguments we provided are also accessible as attributes
  - The documentation page for the resource also lists all the attributes available

The *data* Directive

- References a type of resource that is not under terraform control
  - We supply attributes that are used to identify the specific resource
- In the *main.tf* file we look for a "aws_vpc" where the attribute *default* has the value *true*
  - We need to provide enough information to uniquely identify the resource we are looking for
  - For example, there may be many VPCs in a region, but only one default VPC
- Later we will look at ways of searching for specific resources

```
15
16    data "aws_vpc" "default_VPC" {
17        default = true
18    }
19
```

The *output* Directive

- Returns a value, usually an attribute of AWS resource
  - In the root module, the value is returned to the command line where it is printed out
  - We can also specify an output file where the returned values will be stored
- Including a *description* is considered to be a best practice
- The *value* parameter is what the defined output returns

outputs.tf > ...

```
1
2   output "EC2_public_ip" {
3       description = "Public IP address of 'myVM"
4       value = aws_instance.myVM.public_ip
5   }
6
7   output "VPC_id" {
8       value = data.aws_vpc.default_VPC.id
9   }
10
```

The Terraform Workflow

- Terraform is a declarative language
  - The *.tf source files only describe the final state the AWS environment should be in
  - How to implement the requested state is figured out by terraform
- The basic flow is:
  - Model the desired state of the AWS environment by reading the terraform source code files
  - Fetch a description of the actual AWS environment state
  - Compare the actual and desired states
  - Compute a plan for changing the existing environment so that it conforms to the desired state
  - Create a series of actions to be executed to implement the plan
  - Apply the actions

The Terraform Workflow

- The workflow is implemented through a series of terraform commands
- *init* - scans the source files and updates the local providers
- *validate* - checks for syntax errors in the *.tf files
- *plan* - creates an implementation plan
- *apply* - implements the implementation plan
- *destroy* - removes all AWS resources defined in this module

The Terraform Workflow

- The *apply* command automatically runs *validate* and then *plan*
  - *apply* can also apply a saved plan
- The *plan* command automatically runs *validate*
  - *plan* can also save the plan to a file for later application
- Running *validate* on its own is a lot faster for quick syntax checks

Example - Init Output

Command Prompt

```
C:\home\terraform>terraform init

Initializing the backend...

Initializing provider plugins...
- Finding hashicorp/aws versions matching ">= 3.0.0"...
- Installing hashicorp/aws v3.41.0...
- Installed hashicorp/aws v3.41.0 (signed by HashiCorp)

Terraform has created a lock file .terraform.lock.hcl to record the provider
selections it made above. Include this file in your version control repository
so that Terraform can guarantee to make the same selections by default when
you run "terraform init" in the future.

Terraform has been successfully initialized!

You may now begin working with Terraform. Try running "terraform plan" to see
any changes that are required for your infrastructure. All Terraform commands
should now work.

If you ever set or change modules or backend configuration for Terraform,
rerun this command to reinitialize your working directory. If you forget, other
commands will detect it and remind you to do so if necessary.
```

Example - Plan Output

```
C:\home\terraform>terraform plan -out myplan

Terraform used the selected providers to generate the following execution plan. Resource actions are indicated with the
following symbols:
  + create

Terraform will perform the following actions:

  # aws_instance.myVM will be created
  + resource "aws_instance" "myVM" {
      + ami                                  = "ami-077e31c4939f6a2f3"
      + arn                                  = (known after apply)
      + associate_public_ip_address          = (known after apply)
      + availability_zone                    = (known after apply)
      + cpu_core_count                       = (known after apply)
      + cpu_threads_per_core                 = (known after apply)
```

```
Plan: 2 to add, 0 to change, 0 to destroy.

Changes to Outputs:
  + EC2_public_ip = (known after apply)
  + VPC_id        = "vpc-36ae795d"


Saved the plan to: myplan

To perform exactly these actions, run the following command to apply:
    terraform apply "myplan"
```

Example - Apply Output

```
C:\home\terraform>terraform apply

Terraform used the selected providers to generate the following execution plan. Resource actions are indicated with the
following symbols:
  + create

Terraform will perform the following actions:

  # aws_instance.myVM will be created
  + resource "aws_instance" "myVM" {
      + ami                                  = "ami-077e31c4939f6a2f3"
      + arn                                  = (known after apply)
      + associate_public_ip_address          = (known after apply)
      + availability_zone                    = (known after apply)
```

Example - Apply Output

```
Plan: 2 to add, 0 to change, 0 to destroy.

Changes to Outputs:
  + EC2_public_ip = (known after apply)
  + VPC_id        = "vpc-36ae795d"

Do you want to perform these actions?
  Terraform will perform the actions described above.
  Only 'yes' will be accepted to approve.

  Enter a value: yes

aws_s3_bucket.myBucket: Creating...
aws_instance.myVM: Creating...
aws_s3_bucket.myBucket: Creation complete after 3s [id=terraform-example-02-01]
aws_instance.myVM: Still creating... [10s elapsed]
aws_instance.myVM: Still creating... [20s elapsed]
aws_instance.myVM: Creation complete after 23s [id=i-00e1451d0a26e009e]

Apply complete! Resources: 2 added, 0 changed, 0 destroyed.

Outputs:

EC2_public_ip = "18.116.59.159"
VPC_id = "vpc-36ae795d"
```

Lab 2-1

- Please do Lab 2-1

Terraform Variables

- Variables are used to replace hardcoded values, like the instance type, in terraform code
- Variables can be declared as a specific data type, but default to string
- Variables can have an optional default value
- If a value for a variable is not provided, then the user is prompted to supply the value at the command line when *terraform plan* is run
- Variables are reference by using the syntax:
  - *var.<variable-name>*
  - An older deprecated syntax you might see in legacy code is *${var.<variable-name>}*

Defining Variables

- In the *variables.tf* file, two variable are defined
  - There is a default defined for the *ami_type*
  - The default is used *only* if the variable is not assigned a value anywhere
- The value for the *inst_type* variable is assigned in the *terraform.tfvars* file

variables.tf > ...

```
1
2   variable ami_type {
3       description = "ami to be used in myVM"
4       type = string
5       default = "ami-077e31c4939f6a2f3"
6
7   }
8
9   variable inst_type {
10      description = "instance type for myVM"
11      type = string
12  }
```

terraform.tfvars

```
1   inst_type = "t2.nano"
2
```

Using Variables

- The hardcoded values for the arguments can now be replaced with variables

main.tf > ...

```
1
2   # Example 02-02
3
4   resource "aws_instance" "myVM" {
5       ami = var.ami_type
6       instance_type = var.inst_type
7       tags = {
8           Name = "Example-02"
9       }
10  }
11
```

Output Return Values

- The outputs now validate that the actual attributes of the EC2 were set by the variables

outputs.tf > ...

```
1
2   output "EC2_ami" {
3       description = "ami type used in myVM"
4       value = aws_instance.myVM.ami
5   }
6
7   output "EC2_type" {
8       description = "instance type used in myVM"
9       value = aws_instance.myVM.instance_type
10  }
11
```

Lab 2-2

- Please do Lab 2-2

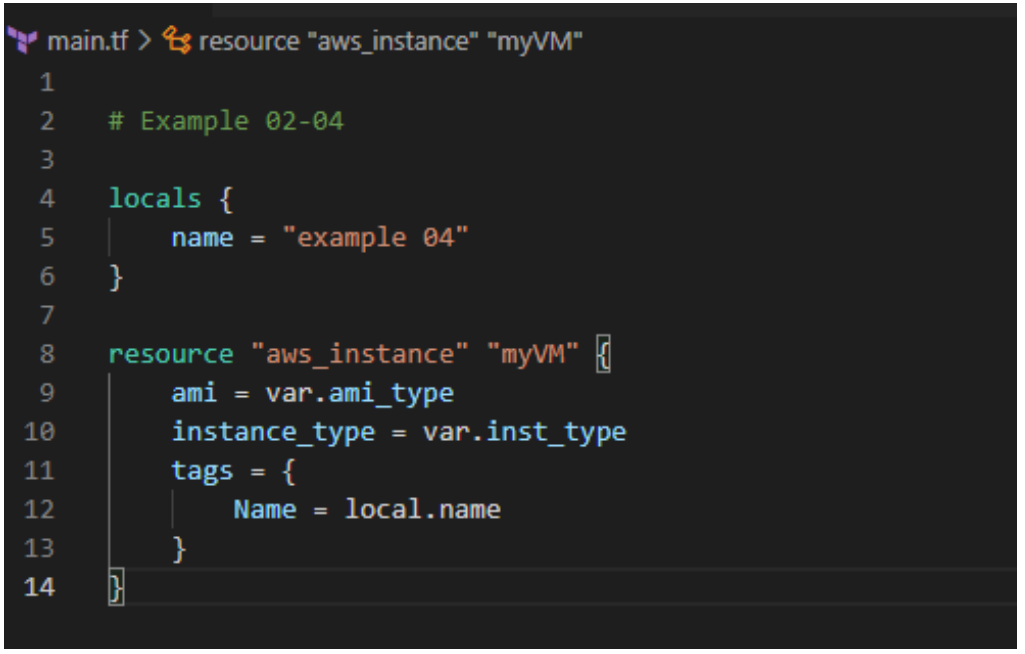
String Interpolation

- Any attribute or value can be embedded in a string by using *string interpolation*
- The interpolation syntax is ${value} to insert "value" into string
- Non-string values are converted to a string for interpolation

```
7   output "EC2_type" {
8       description = "instance type used in myVM"
9       value = "Hi, I am a ${aws_instance.myVM.instance_type} instance type"
10  }
11
```

Local Variables

- Just like in a programming language, we can define local variables that can be used within a module
  - Local variables cannot be referenced outside the module
  - Local variables are defined in a *locals* block
  - Local variable definitions can be split across more than one locals block
  - Local variables are referenced with the syntax *local.<name>*



Primitive Data Types

- Variables and locals are typed data
- There are three primitive data types
  - *string* : A Unicode string
  - *numeric* : Used for both integral and non-integral values (434 and 1.34)
  - *boolean* : `true` and `false`
- There are also complex data types like lists and maps which will be covered in a later module.

```
# Example 02-05

locals {
    name = "example 05"
    port = 8080
    private = true
}
```

Heredoc Strings

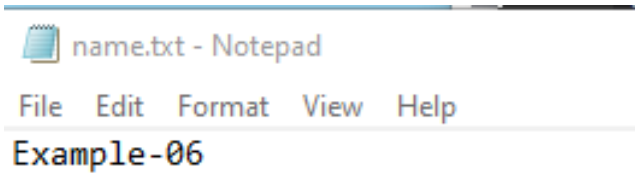
- Terraform has a "heredoc" string literal like , which allows multi-line strings to be expressed more clearly.
- A heredoc string consists of:
  - An opening sequence consisting of:
    - A heredoc marker (<< or <<- — two less-than signs, with an optional hyphen for indented heredocs)
    - A delimiter word of your own choosing
    - A line break
  - The contents of the string, which can span any number of lines
  - The delimiter word you chose, alone on its own line (with indentation allowed for indented heredocs)

```
message = <<-MYSRT
this is a multi line string
that goes on and one and on
MYSRT
```

Reading Files

- Local variables *must* be initialized when defined
  - We cannot set their value with a *terraform.tfvars* entry
- The alternative to hardcoding a local variable is to read its value from a file
  - This is generally done when the variable is used to provide some metadata like the ID of the person running the code
  - Or when the variable contains text like a start-up script
- We use the *file* command to read in the contents of a text file

```
1
2   # Example 02-06
3
4   locals {
5       name = file("name.txt")
6
7   }
```



Lab 2-3

- Please do Lab 2-3

Template Files

- Using files provides some flexibility, but often we want to customize the contents of the file using some terraform variables
- To provide this facility, Terraform has a *templatefile* which
  - Reads in file with "slots" defined using string interpolation syntax
  - Supplies arguments to fill in the slots
- In the example below
  - The template file *document.txt* has a "slot" for "myname" that can be filled in
  - *This code was modified by ${myname}*
  - A local variable called *developer* used to fill in the slot

```
locals {
    developer = "Zippy"
    documentation = templatefile("document.txt", {myname = local.developer})
}

resource "aws_instance" "myVM" {
    ami = var.ami_type
    instance_type = var.inst_type
    tags = {
        Name = "${local.developer}'s machine"
    }
}
```

Rendered Template File

- When all the variables are inserted into the template string, the result is said to be rendered
- The rendered string can be output as shown below

```
output "Documentation" {
    description = "Developer who worked on this"
    value = local.documentation
}
```

Lab 2-4

- Please do Lab 2-4

Data Filters

- Earlier, the *data* construct was introduced as a way to find AWS resources
- A problem that has been sidestepped so far is that the aws_instance code is not portable across regions because the ami instance id's are unique to a region
- To resolve this, a data resource can be used get a specific ami id in a region
  - The problem is trying to find just the one wanted
- The solution is to list all the properties the ami template has and then filter each property to perserve only the ones we want
  - This is not an ad-hoc solution but requires a knowledge of the attributes of the resource to be filtered
  - *And* a specification of which attribute we want to query on

AMI Example

```
# Example 02-08

data "aws_ami" "ubuntu" {
  most_recent = true

  filter {
    name   = "name"
    values = ["ubuntu/images/hvm-ssd/ubuntu-focal-20.04-amd64-server-*"]
  }

  filter {
    name   = "virtualization-type"
    values = ["hvm"]
  }

  owners = ["099720109477"] # Canonical
}
```

```
output "AMI" {
    description = "The Ubuntu AMI"
    value = data.aws_ami.ubuntu.id
}
```

Multiple Providers

- So far, all the code examples have used a default provider, but we can specify another provider by using an alias.
- In this *providers.tf* , the default provider is AWS region *us-east-2* , but a second provider for *us-east-1* can be set with the alias "Virginia"
- We could have also had any other providers, like an Azure provider as well

```
provider aws {
    region = "us-east-2"
    profile = "dev"
}

provider aws {
    region = "us-east-1"
    alias = "Virginia"
    profile = "dev"
}
```

Using Multiple Providers

- For a terraform resource, the *providers* argument specifies which provider will manage the resource
  - In no provider is specified, the default provider is used
- In the example below, the EC2 instance "Ohio" is being created by the default provider while "Virginia" is being created by the us-east-1 provider

```
# Example 02-09

resource "aws_instance" "Ohio" {
    ami = "ami-00399ec92321828f5"
    instance_type = "t2.micro"
    tags = {
        Name = "us-east-2"
    }
}

resource "aws_instance" "Virginia" {
    provider = aws.Virginia
    ami = "ami-09e67e426f25ce0d7"
    instance_type = "t2.micro"
    tags = {
        Name = "us-east-1"
    }
}
```

Lab 2-5

- Please do lab 2-5
- This is a bit more challenging to end the module