04. Terraform
Language and Templates

Contents

- Introduction – Summary of a Terraform Configuration
- Terraform Block
- Provider Block
- Variables: Input, Local, Output
- Resources
- Resource Meta-arguments
- Data Sources
- Terraform Expressions

Terraform File(s) – High Level Overview

- Terraform and Provider(s) Section
  - (optionally) Specify required version of Terraform
  - Specify required providers, their versions and configuration
  - In the examples of this course, you will typically find them in a file called providers.tf (just a convention)
- Resource Section
  - Main purpose of IaC: create resources
  - Contains the definition of the resources we are going to create with Terraform
- Module blocks
  - Used when "calling" Terraform modules defined outside of this directory
- Data source Section
  - Reference resources that exist in the cloud
- Variables Section
  - Input variables (typically in a file called variables.tf)
  - Local variables (typically in the file main.tf or in other .tf files)
  - Output values (typically in a file called outputs.tf)
- Note: it is important to remember all the blocks / sections mentioned above could be in a single .tf file. For the sake of clarity, however, we choose to distribute them among different files (providers.tf, variables.tf, main.tf, network.tf, outputs.tf, etc.)
- Demo – review a largish set of Terraform files

Intro

- This chapter follows roughly the "[Terraform Language](#)" documentation

Terraform Language - Intro

Terraform Language Constructs

- The main purpose of the Terraform language is declaring resources, which represent infrastructure objects.
  - All other language features exist only to make the definition of resources more flexible and convenient.
- A *Terraform configuration* is a complete document in the Terraform language that tells Terraform how to manage a given collection of infrastructure.
  - A configuration can consist of multiple files and directories.
- The syntax of the Terraform language consists of only a few basic elements:
  - Blocks
  - Arguments
  - Expressions

Terraform code: "Blocks"

- Different types of block in Terraform code:
  - Fundamental
    - Terraform Block
    - Providers Block
    - Resources Block
  - Variables
    - Input Variables Block
    - Output Variables Block
    - Local Values Block
  - Calling / Referencing
    - Data Sources Block
    - Modules Block

Terraform Block

terraform block (azure)

- Determines behavior of Terraform and the required providers optionally their version constraints (highly recommended)
- For terraform CLI, determines also the version to be used (e.g. 1.4.1)
- Documentation on version constraints
- Terraform block also includes a section on backends (discussed in chapter on state)
- Running *terraform init* loads the required providers
- Terraform and Providers block often stored in file "providers.tf". This course follows that convention
- (others prefer "versions.tf")

```
terraform {
  required_version = "~>1.2.0"

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

    aws = {
      source = "hashicorp/aws"
      version = ">= 4.25.0"
    }
  }
}

provider "azurerm" {
  features {
    # https://registry.terraform.io/providers/hashicorp/azurerm/latest/docs/guides/features-block
    # virtual_machine_scale_set {
    #   force_delete                 = false
    #   roll_instances_when_required = true
    #   scale_to_zero_before_deletion = true
    # }
  }
  environment = "public"
#   subscription_id = "a1e01a15-61aa-4f25-aa66-6d6e8a913dc3"
}


provider "aws" {
  region = "eu-west-1"
  profile = "tfprofile"
}
```

```
rafa@rp3:01-terraform-provider$ terraform init

Initializing the backend...

Initializing provider plugins...
- Finding hashicorp/azurerm versions matching "~> 3.0"...
- Finding hashicorp/aws versions matching ">= 4.25.0"...
- Installing hashicorp/azurerm v3.18.0...
- Installed hashicorp/azurerm v3.18.0 (signed by HashiCorp)
- Installing hashicorp/aws v4.25.0...
- Installed hashicorp/aws v4.25.0 (signed by HashiCorp)
```

terraform block (aws)

- Determines behavior of Terraform and the required providers optionally their version constraints (highly recommended)

- For terraform CLI, determines also the version to be used (e.g. 1.4.1)

- Documentation on [version constraints](#)

- Terraform block also includes a section on backends (discussed in chapter on state)

- Running *terraform init* loads the required providers

- Terraform and Providers block often stored in file "providers.tf". This course follows that convention

- (others prefer "versions.tf")

```
terraform {
  required_version = "~> 1.4.0"

  required_providers {
    aws = {
      source  = "hashicorp/aws"
      version = "~> 4.13" # v3.38.0 minimal version to use default tags
    }
  }
}


provider "aws" {
  region  = var.region
  profile = var.profile
  default_tags {
    tags = {
      "${var.company}:environment" = var.environment
      "${var.company}:project"     = var.project
      created_by                   = "terraform"
      disposable                   = true
    }
  }
}
```

```
rafa@rp3:01-terraform-provider$ terraform init

Initializing the backend...

Initializing provider plugins...
- Finding hashicorp/azurerm versions matching "~> 3.0"...
- Finding hashicorp/aws versions matching ">= 4.25.0"...
- Installing hashicorp/azurerm v3.18.0...
- Installed hashicorp/azurerm v3.18.0 (signed by HashiCorp)
- Installing hashicorp/aws v4.25.0...
- Installed hashicorp/aws v4.25.0 (signed by HashiCorp)
```

Providers Block

provider block (azure)

- Determines parameters specific to a given provider
- Example shows azurerm and aws providers
- It is also possible to have multiple versions of a provider (e.g. to use *azurerm* provider with two subscriptions, or the *aws* provider in two different regions) -
  - Uses concept of provider alias (example in this [link](#))

```
terraform {
  required_version = "~>1.2.0"

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

    aws = {
      source = "hashicorp/aws"
      version = ">= 4.25.0"
    }
  }
}

provider "azurerm" {
  features {
    # https://registry.terraform.io/providers/hashicorp/azurerm/latest/docs/guides/features-block
    # virtual_machine_scale_set {
    #   force_delete                = false
    #   roll_instances_when_required = true
    #   scale_to_zero_before_deletion = true
    # }
  }
  environment = "public"
#   subscription_id = "a1e01a15-61aa-4f25-aa66-6d6e8a913dc3"
}


provider "aws" {
  region = "eu-west-1"
  profile = "tfprofile"
}
```

```
rafa@rp3:01-terraform-provider$ terraform init

Initializing the backend...

Initializing provider plugins...
- Finding hashicorp/azurerm versions matching "~> 3.0"...
- Finding hashicorp/aws versions matching ">= 4.25.0"...
- Installing hashicorp/azurerm v3.18.0...
- Installed hashicorp/azurerm v3.18.0 (signed by HashiCorp)
- Installing hashicorp/aws v4.25.0...
- Installed hashicorp/aws v4.25.0 (signed by HashiCorp)
```

provider block (aws)

- Determines parameters specific to a given provider
- Example shows azurerm and aws providers
- It is also possible to have multiple versions of a provider (e.g. to use *azurerm* provider with two subscriptions, or the *aws* provider in two different regions) -
  - Uses concept of provider alias (example in this [link](#))

```
terraform {
  required_version = "~> 1.4.0"

  required_providers {
    aws = {
      source  = "hashicorp/aws"
      version = "~> 4.13" # v3.38.0 minimal version to use default tags
    }
  }
}


provider "aws" {
  region  = var.region
  profile = var.profile
  default_tags {
    tags = {
      "${var.company}:environment" = var.environment
      "${var.company}:project"     = var.project
      created_by                   = "terraform"
      disposable                   = true
    }
  }
}
```

Terraform Providers

- Review of previous module from another angle:
  - Terraform – used to manage multiple IaaS (Cloud Providers), PaaS (e.g. Kubernetes) and even SaaS
  - Each of the above environments has "Resources" to be configured and exposes APIs.
  - Terraform/Hashicorp [defines](#) providers as : (...) *a logical abstraction of an upstream API. They are responsible for understanding API interactions and **exposing resources**.*
 - Essentially a provider is a Terraform core plugin

Terraform Registry
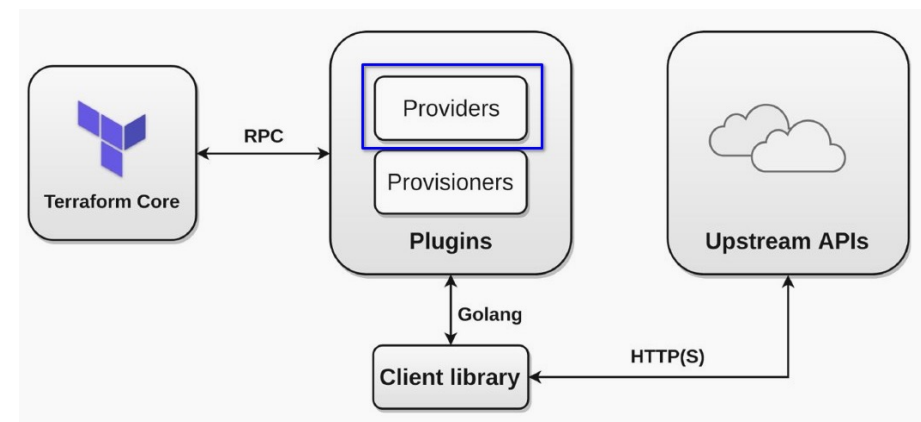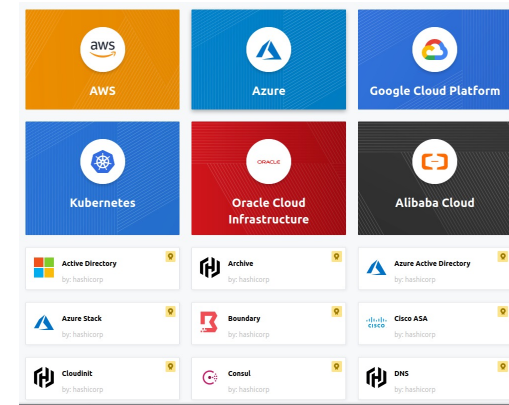
- https://registry.terraform.io/
- Contains Providers and Modules
  - **Providers** are plugins that implement resource types. Find providers for the cloud platforms and services you use, add them to your configuration, then use their resources to provision infrastructure.
  - **Modules** are small, reusable Terraform configurations that let you manage a group of related resources as if they were a single resource.
    - Example: AWS VPC module
    - Example: AzureRM VNet module
  - Evolution:
    - August 2022: 2300+ providers, 10000+ modules & counting
    - March 2023: 2991 providers, 12778 modules & counting

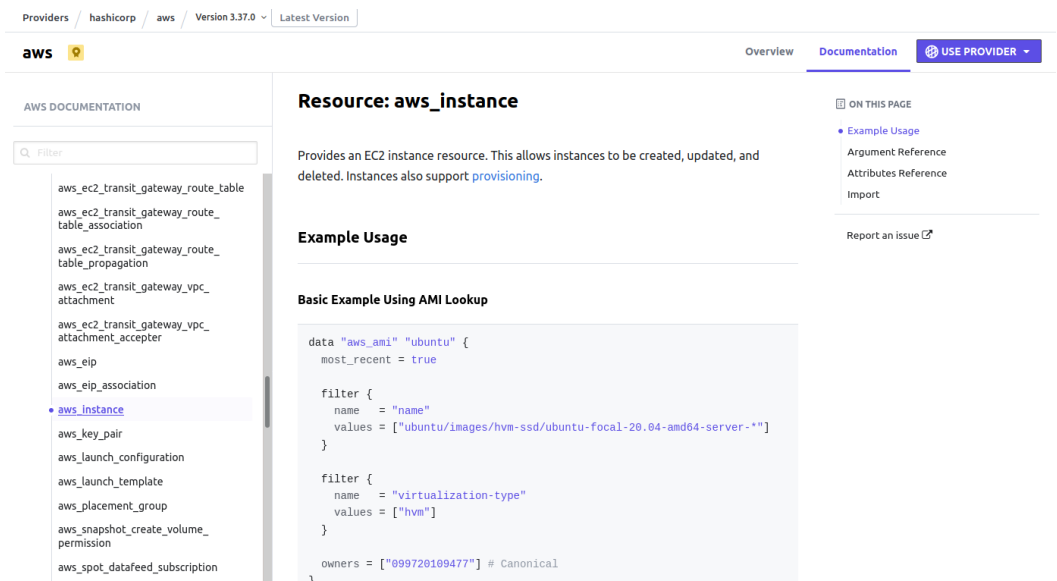- "Featured" providers for major cloud providers

Provider Classification

Tier	Description	Namespace
Official	*Official providers are owned and maintained by HashiCorp*	`hashicorp`
Verified	*Verified providers are owned and maintained by third-party technology partners. Providers in this tier indicate HashiCorp has verified the authenticity of the Provider's publisher, and that the partner is a member of the HashiCorp Technology Partner Program.*	Third-party organization, e.g. `mongodb/mongodbatlas`
Community	Community providers are published to the Terraform Registry by individual maintainers, groups of maintainers, or other members of the Terraform community.	Maintainer's individual or organization account, e.g. `DeviaVir/gsuite`
Archived	Archived Providers are Official or Verified Providers that are no longer maintained by HashiCorp or the community. This may occur if an API is deprecated or interest was low.	`hashicorp` or third-party

AWS Provider Documentation

- Link to AWS provider documentation and source code (golang) in GitHub
- Documents in full detail all AWS Resources modeled / abstracted by Terraform
  - Parameters, output values, etc.
  - Example of resource: AWS EC2 instance (*aws_instance*)

Optimizing disk space - Plugin Cache Directory


```
rafa@rp3:notes$ cat ~/.terraformrc
plugin_cache_dir   = "$HOME/.terraform.d/plugin-cache"
```

- Among the terraform CLI configuration options in file ~/.terraformrc

- If "plugin_cache_dir" is set to a specific directory it enables plugin caching

- From [Documentation](#)

  - By default, *terraform init* downloads plugins into a subdirectory of the working directory so that each working directory is self-contained. As a consequence, if you have multiple configurations that use the same provider then a separate copy of its plugin will be downloaded for each configuration.

  - Given that provider plugins can be quite large (on the order of hundreds of megabytes), this default behavior can be inconvenient for those with slow or metered Internet connections. Therefore Terraform optionally allows the use of a local directory as a shared plugin cache, which then allows each distinct plugin binary to be downloaded only once.

Example - Plugin cache directory

- Example of plugin cache after a few days use of terraform CLI in a linux system.
  - Each time a "provider" statement references a new version, or a new version is available, the relevant plugin is downloaded
- Notes:
  - All plugins currently downloaded / installed are HashiCorp's
  - Multiple versions of azurerm plugin
- Total disk use for **shared** AWS, Azure and GCP plugins ~5GB

```
rafa@rp3:plugin-cache$ du -hd4 . | grep azurerm
165M    ./registry.terraform.io/hashicorp/azurerm/3.17.0
170M    ./registry.terraform.io/hashicorp/azurerm/3.0.2
165M    ./registry.terraform.io/hashicorp/azurerm/3.16.0
167M    ./registry.terraform.io/hashicorp/azurerm/2.86.0
150M    ./registry.terraform.io/hashicorp/azurerm/2.64.0
172M    ./registry.terraform.io/hashicorp/azurerm/2.99.0
165M    ./registry.terraform.io/hashicorp/azurerm/3.14.0
165M    ./registry.terraform.io/hashicorp/azurerm/3.18.0
164M    ./registry.terraform.io/hashicorp/azurerm/3.9.0
164M    ./registry.terraform.io/hashicorp/azurerm/3.13.0
1,7G    ./registry.terraform.io/hashicorp/azurerm
```

More on Plugin Cache Directory

- In a given directory with terraform files that reference the providers, terraform still creates the .terraform directory for the plugins

- Instead of copying (again) the actual plugins to this directory, it places symbolic links to actual provider files under the plugin cache directory

```
terraform {
  required_version = "~>1.2.0"

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

    aws = {
      source  = "hashicorp/aws"
      version = ">= 4.25.0"
    }
  }
}
```

```
rafa@rp3:01-terraform-provider$ terraform init

Initializing the backend...

Initializing provider plugins...
- Finding hashicorp/azurerm versions matching "~> 3.0"...
- Finding hashicorp/aws versions matching ">= 4.25.0"...
- Installing hashicorp/azurerm v3.18.0...
- Installed hashicorp/azurerm v3.18.0 (signed by HashiCorp)
- Installing hashicorp/aws v4.25.0...
- Installed hashicorp/aws v4.25.0 (signed by HashiCorp)
```

```
rafa@rp3:01-terraform-provider$ tree .terraform
.terraform
└── providers
    └── registry.terraform.io
        └── hashicorp
            ├── aws
            │   └── 4.25.0
            │       └── linux_amd64 -> /home/rafa/.terraform.d/plugin-cache/registry.terraform.io/hashicorp/aws/4.25.0/linux_amd64
            └── azurerm
                └── 3.18.0
                    └── linux_amd64 -> /home/rafa/.terraform.d/plugin-cache/registry.terraform.io/hashicorp/azurerm/3.18.0/linux_amd64
```

Dependency Lock File

- Feature introduced in release 0.14

- Simplified version of purpose of this file: help use the same provider version even if a new one is available, unless we explicitly require it in the configuration

- e.g. if we specify for azurerm provider : version = ">= 3.10.0"
  - The first time we run terraform init the latest version of provider satisfying this constraint will be downloaded (e.g 3.12.0). This info (and checksum of provider) will be stored in the file .terraform.lock.hcl
  - The next time we run terraform init, terraform will check the depenency lock file and will use the version state there (3.12.0), and even if version 3.15.0 is available it will not download/use it. This ensures a consistent environment.

- TF Documentation

- TF Tutorial (uses AWS)

- Terraform recommendation: include the lock file in your version control repository to ensure that Terraform uses the same provider versions across your team and in ephemeral remote execution environments.

- Related command: terraform init –upgrade

- Example : file in course repo: 04-terraform-language/01-terraform-provider/example.dependency.lock.md

Terraform Resources

Terraform Resources (aws)

- https://www.terraform.io/docs/language/resources/index.html
- *Resources* are the most important element in the Terraform language.
- Each resource block describes one or more infrastructure objects, such as virtual networks, virtual machines, or higher-level components such as DNS records.
- A terraform configuration file is "mainly" based on a series of **linked** resources
  - Example: a storage container definition links to a storage account, that in turn links to a resource group
- Demo: explore the structure of azurerm provider for resource documentation using azurerm_storage_container as an example
- Exercise: practice with the VS Code Terraform extension (hint: "Ctrl-Space")

```
resource "aws_instance" "server1" {
  ami           = var.my_ami
  instance_type = var.instance_type

  vpc_security_group_ids = [aws_security_group.sec_web.id]

  tags = {
    Name    = "vm_lab2"
    project = var.project
  }
}
```

Terraform Variables

Terraform Variables

- Terraform [Documentation on Variables](#)
- Three types of variables in a Terraform configuration
  - [Input Variables](#) serve as parameters for a Terraform module, so users can customize behavior without editing the source.
    - Similar to input variables to a programming language function
    - When we talk about "variables" in terraform we are usually referring to input variables
  - [Output Values](#) are like return values for a Terraform module.
    - Similar to return values of a programming language function
  - [Local Values](#) are a convenience feature for assigning a short name to an expression.
    - Similar to local internal values inside a programming language function

Input Variables

Terraform (Input) Variables

- [Terraform Documentation](#)

- In Terraform documentation and code, when referring to "variables" we generally mean "input variables"

- Input variables let you customize aspects of Terraform modules without altering the module's own source code.

- This allows you to share modules across different Terraform configurations, making your module composable and reusable.

- When you declare variables in the root module of your configuration, you can set their values using CLI options and environment variables.

- **NOTE**: we will revisit input variables in the course section on TF modules

Variable Definition / Use (aws)

- Definition (convention: variables.tf file)
  - Variable definitions cannot refer to other variables (one more reason to use locals)
- Use in a resource definition using notation *"var.<variable_name>"*
- Variable Validation (introduced in 0.13) – implement rules (e.g. "var.env" must be "prod" or "staging")
  - TF Docs
  - Third party blogs – examples
  - Note function can() - "designed" mainly for this purpose

```
variable "environment" {
  type        = string
  default     = "dev"
  description = "Workload environment"

  validation {
    condition     = can(regex("^dev$|^prod$|^test$", var.environment))
    error_message = "Err: invalid environment."
  }

  validation {
    condition     = length(var.environment) <= 4
    error_message = "Err: environment is too long."
  }
}
```

```
variable "vpc_cidr" {
  type    = string
  default = "10.99.0.0/16"
  validation {
    condition     = can(cidrnetmask(var.vpc_cidr))
    error_message = "Invalid CIDR for VPC."
  }
}
```

Asigning Values to Variables

```
terraform.tfvars > ...
  1   region = "eastus"
  2   environment = "dev"
  3   cost_center = "PI-31416"
  4   vmsize = "Standard_DS1_v2"
  5
  6   |
```

- TF Docs
- CLI: Individually, with the -var command line option.
  - terraform plan –var="region=westeurope" (azure)
  - terraform plan –var="region=eu-south-2" (aws)
  - See examples for more complex expressions including for windows
- In variable definitions (.tfvars) files, either specified on the command line or automatically loaded.
  - terraform apply –varfile="somefile.tfvars"
  - Some file names loaded automatically
    - Files named exactly terraform.tfvars or terraform.tfvars.json.
    - Files with names ending in .auto.tfvars or .auto.tfvars.json
- As environment variables.
  - export TF_VAR_region=eu-south-2 (linux)
  - terraform apply
- "default" in variable definition

- Precedence:
  - Environment variables
  - The terraform.tfvars file, if present.
  - The terraform.tfvars.json file, if present.
  - Any *.auto.tfvars or *.auto.tfvars.json files, processed in lexical order of their filenames.
  - Any -var and -var-file options on the command line, in the order they are provided. (This includes variables set by a Terraform Cloud workspace.)

Lab – Variables (aws)

- Based on lab_02_ec2 configuration from repo
- Explore different ways to assign variable values, e.g. for "project", "company", "special_port" etc
  - Env Variables
  - CLI parameters to terraform command
  - terraform.tfvars file
- In most the cases, it will simply modify the resource, not re-create it.

Local Variables

Local Values (aws)

- A local value assigns a name to an expression, so you can use it multiple times within a module without repeating it.

- Continuing the analogy with programming languages
  - Input variables are like function arguments.
  - Output values are like function return values.
  - Local values are like a function's temporary local variables.

- A very typical use case: common tags for all created resources: "local.tags" encapsulates a large and potentially messy map that is used in many resources (vpc, alb, etc.)
  - Note: Above still useful in general. but for the AWS provider, since version 3.38.0 there is a new way to add tags to all resources at provider level

- Note: example on the right was very typical. More recently a simpler way to do this in AWS is to use   "default_tags" in the aws provider block

Define in main.tf

```
# local for tag population
locals {
  required_tags = {
    project     = var.project_name
    environment = var.project_env
    disposable  = true
    terraform   = true
  }
  tags = merge(var.resource_tags, local.required_tags)
}
```

Use in vpc.tf

```
module "vpc-webapp" {
  source  = "terraform-aws-modules/vpc/aws"
  version = "2.78.0"
(...)
  tags = local.tags
(...)
```

Use in alb.tf

```
module "alb-webapp" {
  source  = "terraform-aws-modules/alb/aws"
  version = "5.16.0"
  name    = "alb-${var.project_name}-${var.project_env}"

  tags    = local.tags
```

Local Values - Other Useful Examples

- Avoid repeating complex expressions

```
locals {
  name_suffix = "${var.resource_tags["project"]}-${var.resource_tags["environment"]}"
}
```

```
-  name = "vpc-${var.resource_tags["project"]}-${var.resource_tags["environment"]}"
+  name = "vpc-${local.name_suffix}"
```

- Unlike variable values, local values can use dynamic expressions (e.g. conditionals), resource arguments and Terraform functions.

```
locals {
  name_prefix = "${var.project_name == "" ? "NA" : var.project_name }"
}
```

```
locals {
  # Ids for multiple sets of EC2 instances, merged together
  instance_ids = concat(aws_instance.blue.*.id, aws_instance.green.*.id)
}
```

Local Values – Best practices

- Terraform Recommendation
  - Local values can be helpful to avoid repeating the same values or expressions multiple times in a configuration, but if overused they can also make a configuration hard to read by future maintainers by hiding the actual values used.
  - **Use local values only in moderation**, in situations where a single value or result is used in many places *and* that value is likely to be changed in future. The ability to easily change the value in a central place is the key advantage of local values.
- Local values and debugging
  - Useful when using template_file function and user_data

Output Values

Terraform output values

- [Docs](#)
- Output values make information about your infrastructure available on the command line, and can expose information for other Terraform configurations to use.
- Each output refers to a specific value
- Output values are similar to return values in programming languages.
- Common convention: outputs.tf

```
outputs.tf > ...
  1   output "resource_group_name" {
  2     description = "Name of resource group"
  3     value = azurerm_resource_group.rg.name
  4   }
  5
  6   output "public_ip" {
  7     description = "Public IP assigned to the VM"
  8     value = azurerm_linux_virtual_machine.tfub_vm.public_ip_address
  9   }
 10
 11   output "private_ip" {
 12     description = "Private IP assigned to the VM"
 13     value = azurerm_linux_virtual_machine.tfub_vm.private_ip_address
 14   }
 15
 16   output "full_machine_name" {
 17     description = "FQDN of the Public IP assigned to the VM"
 18     value = azurerm_public_ip.tfub_publicip.fqdn
 19   }
 20
 21   output "vm_admin_user" {
 22     description = "Admin user of Linux VM"
 23     value = azurerm_linux_virtual_machine.tfub_vm.admin_username
 24   }
 25
 26   output "name_suffix" {
 27     description = "Comon suffix used for resource naming"
 28     value = local.name_suffix
 29   }
```

terraform output

- Display output values
  - Reading output values from state file is discouraged (considered internal API)
- By default individual outputs are quoted (since around TF 0.13)
- For post processing, use json output and filters (e.g. with jq). Simple example below.
- Convenient way to ping or RDP/SSH to VMs.

```
rafa@rp3:basic-ubuntu-vm$ terraform output
public_ip_address = "52.188.53.45"
resource_group_name = "rg-active-crane"
rafa@rp3:basic-ubuntu-vm$ terraform output -json
{
  "public_ip_address": {
    "sensitive": false,
    "type": "string",
    "value": "52.188.53.45"
  },
  "resource_group_name": {
    "sensitive": false,
    "type": "string",
    "value": "rg-active-crane"
  }
}
rafa@rp3:basic-ubuntu-vm$ terraform output -json | jq -r .public_ip_address.value
52.188.53.45
rafa@rp3:basic-ubuntu-vm$ ping $(terraform output -json | jq -r .public_ip_address.value)
PING 52.188.53.45 (52.188.53.45) 56(84) bytes of data.
64 bytes from 52.188.53.45: icmp_seq=1 ttl=50 time=92.8 ms
64 bytes from 52.188.53.45: icmp_seq=2 ttl=50 time=92.0 ms
```

```
# Using terraform output to automate access to VM
KEY_NAME=mykey
PUB_IP=$(terraform output  -json | jq -r .public_ip.value)
AZ_USER=$(terraform output  -json | jq -r .vm_admin_user.value)
echo "connecting to: $PUB_IP as $AZ_USER"
echo "with key $KEY_NAME"
ssh -i ~/.ssh/$KEY_NAME $AZ_USER@$PUB_IP
```

Resource meta_arguments

Resource Meta-Arguments

- Meta-Arguments can be added to resources to modify their behavior (e.g. specify dependencies, create more instances of the resource, prevent resource from being destroyed)
  - depends_on
  - count
  - for_each
  - provider
  - life_cycle

count and conditionals (azure)

- Meta-argument *count* can also be used to perform conditional deployment of resources

```
resource "azurerm_resource_group" "rg-test" {
  count    = 3
  location = "westeurope"
  name     = "rg-test-${count.index}"
}

output "resource_groups" {
  value = azurerm_resource_group.rg-test
}

output "resource_group_names" {
  value = azurerm_resource_group.rg-test[*].name
}
```

```
rafa@rp3:01-count-rg$ terraform state list
azurerm_resource_group.rg-test[0]
azurerm_resource_group.rg-test[1]
azurerm_resource_group.rg-test[2]
```

```
rafa@rp3:01-count-rg$ terraform output resource_group_names
[
  "rg-test-0",
  "rg-test-1",
  "rg-test-2",
]
```

Meta-Argument: for_each (azure)

- TF Docs
- If a resource or module block includes
  a for_each argument whose value is a **map** or a
  set of strings, Terraform will create one instance
  for each member of that map or set.

- Example: repo 04-terraform-language/03-met...
  arg-count/02-foreach-rg
  - Cannot use splat expressions. Use "for" expression
    instead (next slide)

- Note limitations on values mentioned in
  documentation

Demo lab : *for* expression

- See lab – repo: 20-terraform-
  console/01-console-for-expr

```
1   variable "users" {
2     type = map(object(
3       {
4         dept = string
5         is_admin = bool
6       }
7     ))
8     default = {
9       rafa = {
10        dept = "IT"
11        is_admin = true
12        employee_id = 33333
13      },
14      paco = {
15        dept = "Planning"
16        is_admin = false
17      },
18      "jose andres" = {
19        dept = "Accounting"
20        is_admin = false
21      }
22    }
23  }
24
25  locals {
26    admin_users = {
27      for name, user in var.users : name => user
28      if user.is_admin
29    }
30    regular_users = {
31      for name, user in var.users : name => user
32      if !user.is_admin
33    }
34  }
35
```

for_each – example (azure)

- repo 04-terraform-language/03-meta-arg-count/03-foreach-example
- Create resources across "environments" (dev, prod, etc.)
  - Resource group, storage account, storage containers
- Some important functions covered:
  - merge() for tags

```
variable "all_environments" {
    type = list(string)
    default = ["dev", "prod"]
}
```

```
locals {
#   name_suffix = "${var.app_name}-${var.environment}-${var.region}"
  tags = {
    Source          = "terraform"
    # Env            = var.environment
    CostCenter      = var.cost_center
    ApplicationName = var.app_name
  }
}

resource "azurerm_resource_group" "rg_env" {
    for_each = toset(var.all_environments)
        name = join("-", ["rg", var.app_name, each.value, var.region])
        location = var.region
        tags = merge(
            local.tags,
            {
            "Env" = each.value
            }
        )
    }

output "rg_names" {
    value = {
        for k,v in azurerm_resource_group.rg_env: k => v.name
    }
}
```

```
rafa@rp3:02-foreach-complex$ terraform state list
azurerm_resource_group.rg_env["dev"]
azurerm_resource_group.rg_env["prod"]
```

```
rafa@rp3:02-foreach-complex$ terraform output rg_names
{
  "dev" = "rg-foreach-example-dev-westeurope"
  "prod" = "rg-foreach-example-prod-westeurope"
}
```

Meta-Argument - Resource life_cycle

- TF docs and tutorial with ec2 instance and security group
- Lifecycle arguments help control the flow of your Terraform operations by creating custom rules for resource creation and destruction
- These are:
  - create_before_destroy ->
    - protect against race conditions in creation and destruction of resources
    - demo in backends
  - prevent_destroy ->
    - Protect from accidental destruction
    - demo in backend
  - ignore_changes
    - Prevent TF from "fixing" changes made outside terraform (e.g. tags)

```
# Lifecycle Changes
lifecycle {
    create_before_destroy = true
}
```

```
# Lifecycle Changes
lifecycle {
    prevent_destroy = true
}
```

```
# Lifecycle Changes
lifecycle {
    ignore_changes = [
        tags,
    ]
}
```

Data Sources

Data Sources and AWS AMIs

- Use region-specific AWS AMIs without complicated maps
- Example below for "most recent" Ubuntu 20.04 AMI
- Note: filter must return a single value – otherwise error
- See lab_05 for example

```
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"]
  }

  filter {
    name   = "root-device-type"
    values = ["ebs"]
  }

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

```
resource "aws_instance" "ubusrv02" {
  ami                         = data.aws_ami.ubuntu.id
  instance_type               = var.ec2_instance_type
  associate_public_ip_address = true
  key_name                    = var.ec2_key_name
}
```

Example: data source for AWS AZs & locals

- Data source for Availability Zones

- Locals to simplify expressions in instance *test0 – local.aznames*

- Locals particularly useful if the value(s) will be used in other places

- Is it worth using locals in this case? Probably yes– for testing and troubleshooting
  - Local variables are useful to verify the values of the AZs before creating any instance
  - Useful in combination with *terraform console*

```
data "aws_availability_zones" "available" {
  state = "available"
}
locals {
  aznames = data.aws_availability_zones.available.names
}

resource "aws_instance" "test0" {
  availability_zone           = local.aznames[0]
  ami                         = data.aws_ami.ubuntu.id
  instance_type               = var.ec2_instance_type
  associate_public_ip_address = true
  key_name                    = "bastion1ireland"
  tags = {
    Name = "ec2-${local.aznames[0]}"
  }
}


resource "aws_instance" "test1" {
  availability_zone           = data.aws_availability_zones.available.names[1]
  ami                         = data.aws_ami.ubuntu.id
  instance_type               = var.ec2_instance_type
  associate_public_ip_address = true
  key_name                    = "bastion1ireland"
  tags = {
    Name = "ec2-${data.aws_availability_zones.available.names[1]}"
  }
}
```

Review lab_02: data source to discover VPCs

- Many data sources for VPC and subnet related constructs in the documentation

- To gather info about default vpc (lab 01 – then used for SG)

- To discover a specific VPC or the default VPC

- To discover subnets and AZs

Local-only Data Sources

- While many data sources correspond to an infrastructure object type that is accessed via a remote network API, some specialized data sources operate only within Terraform itself, calculating some results and exposing them for use elsewhere.

- For example, local-only data sources exist for [rendering templates](#), [reading local files](#), and [rendering AWS IAM policies](#).

- The behavior of local-only data sources is the same as all other data sources, but their result data exists only temporarily during a Terraform operation, and is re-calculated each time a new plan is created.

Terraform Expressions

Terraform Expressions - Intro

- Docs: Expressions

- HashiCorp Tutorial "Create Dynamic Expressions"

- Contents
  - Types and Values documents the data types that Terraform expressions can resolve to, and the literal syntaxes for values of those types.
  - Strings and Templates documents the syntaxes for string literals, including interpolation sequences and template directives.
  - References to Values documents how to refer to named values like variables and resource attributes.
  - Operators documents the arithmetic, comparison, and logical operators.
  - Function Calls documents the syntax for calling Terraform's built-in functions.
  - Conditional Expressions documents the <CONDITION> ? <TRUE VAL> : <FALSE VAL> expression, which chooses between two values based on a bool condition.
  - For Expressions documents expressions like [for s in var.list : upper(s)], which can transform a complex type value into another complex type value.
  - Splat Expressions documents expressions like var.list[*].id, which can extract simpler collections from more complicated expressions.
  - Dynamic Blocks documents a way to create multiple repeatable nested blocks within a resource or other construct.
  - Type Constraints documents the syntax for referring to a type, rather than a value of that type. Input variables expect this syntax in their type argument.
  - Version Constraints documents the syntax of special strings that define a set of allowed software versions. Terraform uses version constraints in several places.

Terraform Functions

Terraform Functions - General Info

- Terraform [docs](#) (~120 functions as of version 1.4.0)
- Functions can be called from within expressions to transform and combine values.
  - Syntax similar to many programming languages: e.g. *max(5, 12, 9)*   or *zipmap(keyslist, valueslist)*
- No user defined functions in terraform
- Very useful: Experiment with functions in terraform console – no need to apply in a template
  - Recommended – for these tests use terraform console in an independent **local** directory to avoid locking the shared state (more in the managing state section)

```
> zipmap(["a", "b"], [1, 2])
{
  "a" = 1,
  "b" = 2,
}
```

```
> element([0,1,2,3,4,5,6,7,8,9], 2)
2
> element([0,1,2,3,4,5,6,7,8,9], 212)
2
> element([0,1,2,3,4,5,6,7,8,9], 213)
3
```

A (subjective and growing) list of useful functions –

- To make code more robust
  - can
  - coalesce
  - try
- To generate objects and output
  - zipmap
- IP manipulation
  - cidrxxx

can function

- "can" function and variable validation (in itself a very useful functionality)
  - *can* evaluates the given expression and returns a boolean value indicating whether the expression produced a result without any errors.

```
variable "timestamp" {
  type        = string

  validation {
    # formatdate fails if the second argument is not a valid timestamp
    condition     = can(formatdate("", var.timestamp))
    error_message = "The timestamp argument requires a valid RFC 3339 timestamp."
  }
}
```

try function

- [try function docs](#)
- try evaluates all of its argument expressions in turn and returns the result of the first one that does not produce any errors.
- Very useful, but the docs state multiple caveats against overuse, including :
- **Warning:** The try function is intended only for concise testing of the presence of and types of object attributes. Although it can technically accept any sort of expression, **we recommend using it only with simple attribute references and type conversion functions as shown in the examples above**. Overuse of try to suppress errors will lead to a configuration that is hard to understand and maintain.

zipmap

- [Terraform zipmap doc](#)

- Very handy function – see examples =>

- Also very useful to zip together two "splats" when creating multiple resources (in the example below, multiple EC2 instances)

```
> zipmap(["a","b"], [1,2])
{
  "a" = 1
  "b" = 2
}
```

```
> var.public_subnets
tolist([
  "10.0.1.0/24",
  "10.0.2.0/24",
])
> local.azs
[
  "eu-west-1a",
  "eu-west-1b",
]
> zipmap(local.azs, var.public_subnets)
tomap({
  "eu-west-1a" = "10.0.1.0/24"
  "eu-west-1b" = "10.0.2.0/24"
})
```

```
rafa@rp3:tf_count$ echo "zipmap(aws_instance.test.*.id, aws_instance.test.*.public_ip)" | terraform console
{
  "i-004f1b440f9cfe3f0" = "54.229.75.153"
  "i-085ad3946e2a4b548" = "3.249.167.112"
}
```

IP subnet / CIDR manipulation

- **cidrsubnets** : calculates a sequence of consecutive IP address ranges (subnets) within a particular CIDR prefix

- **cidrsubnet** : calculates a subnet address within given IP network address prefix.

- In the first example, we "extract" the first 4 x /24 subnets (8 additional bits) from the 10.44.0.0/16 CIDR: 10.44.0.0/24, 10.44.1.0/24, etc. We then use them as inputs for the subnets of a VPC (modules are discussed later in the course)

- In a second example, we use **cidrsubnet** (note no "s") to get the second half of a /16 and from that second half (which is a /17) we use **cidrsubnets** to extract 2 x /23

- These and most TF functions can be tested with simple configuration files using Terraform console **without having to build any infrastructure**

```
## VPC
variable "vpc_cidr" {
  description = "CIDR for deployment VPC"
  type        = string
  default     = 10.44.0.0/16
}
```

```
locals {
  two_pub_subnets  = slice(cidrsubnets(var.vpc_cidr, 8, 8, 8, 8), 0, 2)
  two_priv_subnets = slice(cidrsubnets(var.vpc_cidr, 8, 8, 8, 8), 2, 4)
}
```

```
module "vpc" {
  source  = "terraform-aws-modules/vpc/aws"
  version = "~> 3.0"
  (...)
  public_subnets  = local.two_pub_subnets
  private_subnets = local.two_priv_subnets
}
```

```
rafa@rp3:tree_tier_aws_modules$ echo "local.two_priv_subnets" | terraform console
tolist([
  "10.44.2.0/24",
  "10.44.3.0/24",
])
rafa@rp3:tree_tier_aws_modules$ echo "local.two_priv_subnets[0]" | terraform console
"10.44.2.0/24"
```

```
rafa@rp3:tree_tier_aws_modules$ echo "cidrsubnet(var.vpc_cidr, 1,1)" | terraform console
"10.44.128.0/17"
rafa@rp3:tree_tier_aws_modules$ echo "cidrsubnets(cidrsubnet(var.vpc_cidr, 1,1), 6,6)" | terraform console
tolist([
  "10.44.128.0/23",
  "10.44.130.0/23",
])
```

Odds and ends

- element(list, index) -
  - Returns a single element from a list at the given index. If the index is greater than the number of elements, this function will wrap using a standard mod algorithm. This function only works on flat lists.
  - Terraform recommends using it only when the wrap-around behavior is needed. Otherwise use *listname[index]* to retrieve an element

  - May be useful to assign subnets to AZs without actually performing modulo operations

```
> element([0,1,2,3,4,5,6,7,8,9], 2)
2
> element([0,1,2,3,4,5,6,7,8,9], 212)
2
> element([0,1,2,3,4,5,6,7,8,9], 213)
3
```