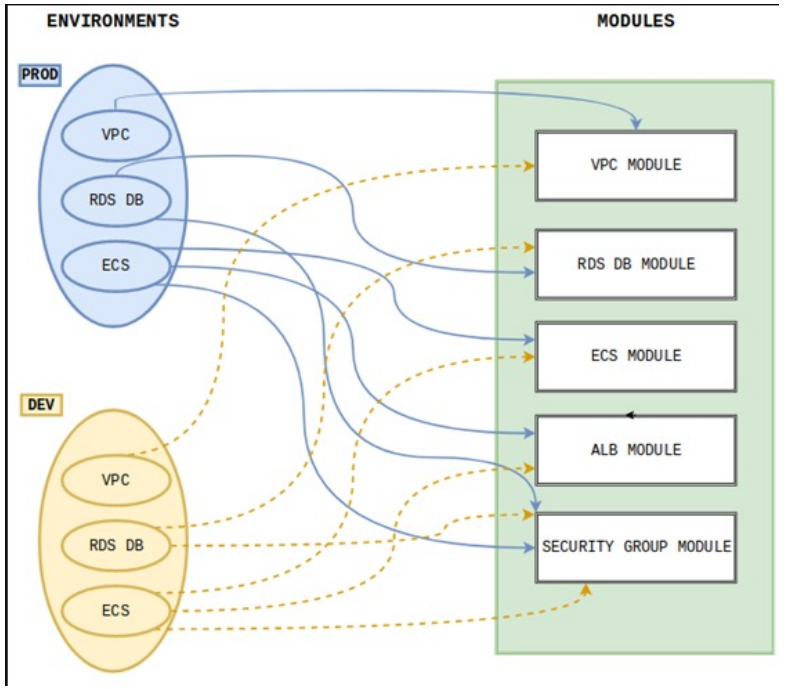
06.Terraform Modules

Contents

- Duplicating code with shareable modules
  - Using Modules
  - Creating Modules
- Using the Module Registry to build reusable templates

Why Modules

- In one sentence: Reuse functionality
- As infrastructure projects grow large and more complex, we need a solution to the problems posed by massive monolithic single-file or single-directory Terraform Code
  - Help understanding and navigating the code
  - Avoid duplication (DRY principle – Do not Repeat Yourself)
  - Control of updates and code changes (e.g. do not update KMS TF code in 20 places...)
  - Share Terraform code among different organizations beyond copy pasting blocks of codes
- Main topics of this section of the course:
  - Using modules created and published by others
  - Writing your own modules (and knowing when to write one...)
  - Module Sources – e.g. from github



Intro to Terraform Modules

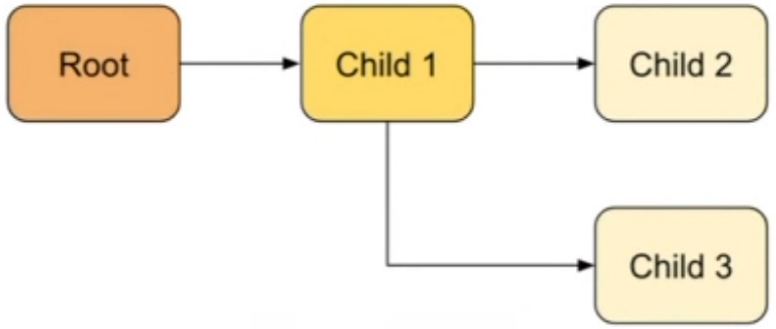
- Terraform docs: https://www.terraform.io/language/modules
- *Modules* are containers for multiple resources that are used together. A module consists of a collection of terraform files kept together in a directory.
- Modules are the main way to **package** and **reuse** resource configurations with Terraform.
  - Typical example: Azure Vnet module to avoid writing "virtual network, subnet, route table, etc. code" all over the place.
- Using a programming language analogy, modules are like libraries or packages that encapsulate and allow the re-use of functionality

Root and Child Modules

- A module is basically a set of .tf files in a directory. Two types:
- Root Module
  - Every Terraform configuration has at least one module, known as its *root module*, which consists of the resources defined in the .tf files in the main working directory.
  - If you have used Terraform, you have been using the "root module" all along …
- Child Modules
  - A Terraform module (usually the root module of a configuration) can ***call*** other modules to include their resources into the configuration. A module that has been called by another module is often referred to as a *child module.*
  - Child modules can be called multiple times within the same configuration, and multiple configurations can use the same child module.

main.tf

```
module "child-1" {
  ...
}
```



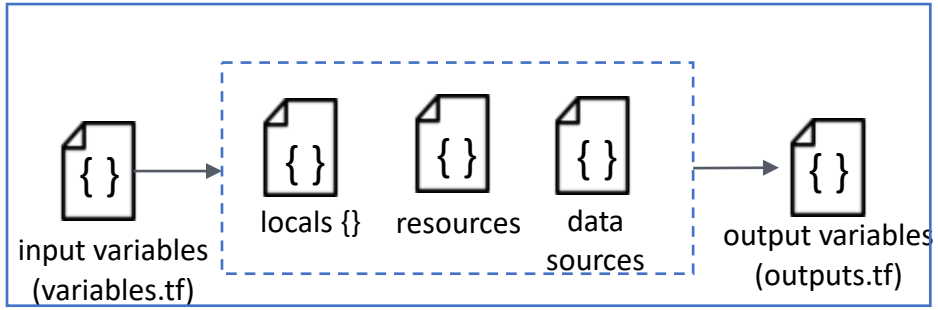
child-1/main.tf

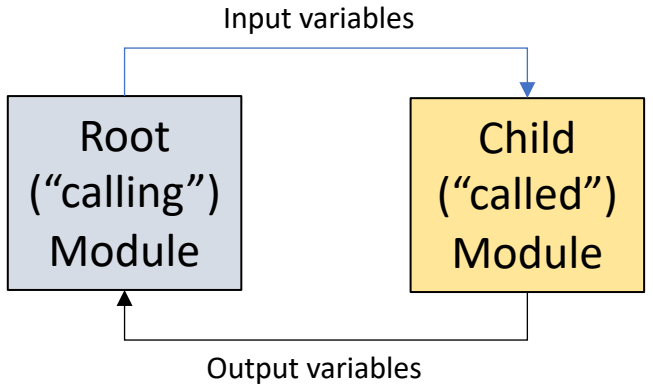
```
module "child-2" {
  ...
}

module "child-3" {
  ...
}
```

Modules and Variables

- Variables are key to write useful TF code
- We use variables in the root module (as seen so far in the course)
- Modules will use variables to communicate
  - Input variables: Provide parameters to the“called” module
  - Output variables: Used by the module to provide results to the “calling” module
- We also use local variables inside the modules



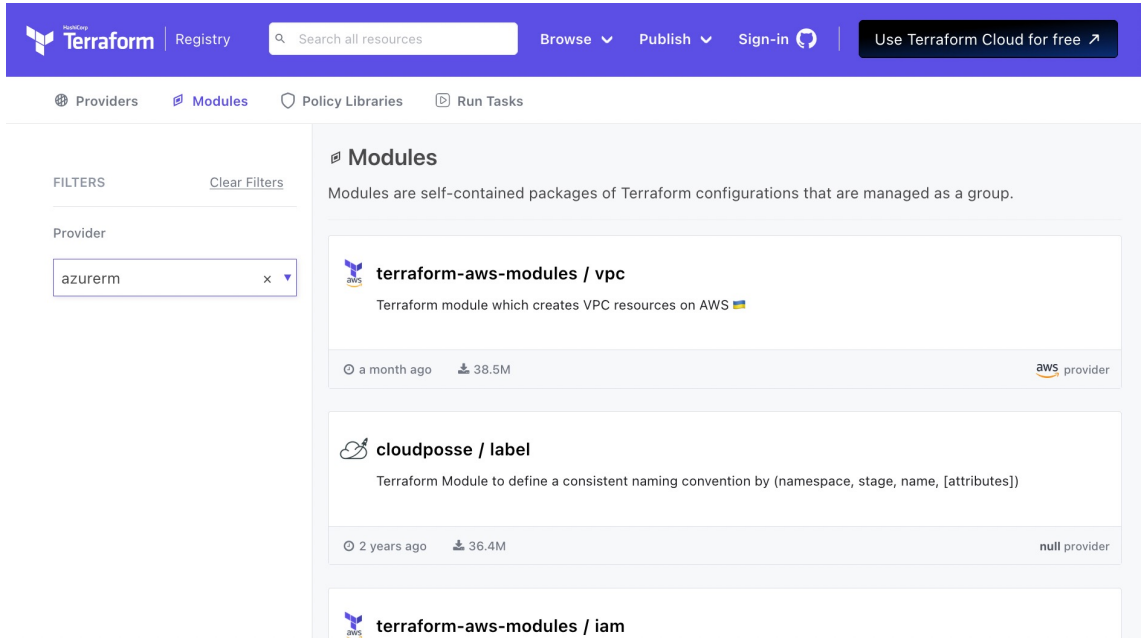


Using 3rd party Modules

- In addition to modules from the local filesystem, Terraform can load modules from a public or private registry. This makes it possible to publish modules for others to use, and to use modules that others have published.
- The Terraform Registry hosts a broad collection of publicly available Terraform modules for configuring many kinds of common infrastructure.
  - These modules are free to use and Terraform can download them automatically if you specify the appropriate source and version in a module call block.
- Also, members of your organization might produce modules specifically crafted for your own infrastructure needs.
  - Terraform Cloud and Terraform Enterprise both include a private module registry for sharing modules internally within your organization.

Exploring 3rd party modules – Module Registry

- Modules are self-contained packages of Terraform configurations that are managed as a group.
- Example
  - Terraform AWS Modules in Registry
    - As of March 2023, 45 Community-supported modules
    - Github Repos
    - Exercise during course – Review and use
      - AWS VPC
      - AWS Security Group



Module Sources - overview

- **source** = where Terraform loads the contents of a module
- Many options are available. See TF Module source docs
  - Terraform Registry (e.g. used for aws vpc module)
  - Local paths ("../../modules/static-website")
  - GitHub (alternative to local paths)
  - Bitbucket
  - Generic Git, Mercurial repositories
  - HTTP URLs
  - S3 buckets
  - GCS buckets
  - Modules in Package Sub-directories

Example / lab – Using the AWS VPC Module

- Review aws vpc module
  - Inputs – variables
  - Outputs
  - Created resources
- Figure shows creation of two VPCs.
  - Two “calls” to module
  - First case : Creates 2 public and 2 private subnets in two AZs. Parameters obtained from variables
  - 2nd Case: Creates 3 public subnets in 3 Azs. Parameters directly from strings
- lab_09a in repo
- Suggested tasks (see README.md)
  - Try changing the name of the module "call" (from ”otra_vpc" to something else) -- what happens?
  - Create a third VPC for example “vpc3” with 2 private subnets. Create an Amazon Linux instance in each of the two subnets using ”count”
  - Explore state and resources created – where do they come from ?
  - Review source code of module (github link in page)
  - Explore the outputs

```
module "vpc_one" {
  source  = "terraform-aws-modules/vpc/aws"
  version = "~> 3.1"

  name               = "vpc-${local.name_suffix}-1"
  cidr               = var.vpc_cidr1
  enable_nat_gateway = false
  single_nat_gateway = true

  azs             = ["${var.region}a", "${var.region}b"]
  public_subnets  = var.pub_subnets
  private_subnets = var.priv_subnets

}

module "otra_vpc" {
  source  = "terraform-aws-modules/vpc/aws"
  version = "~> 3.1"

  name               = "vpc-${local.name_suffix}-2"
  cidr               = "192.168.0.0/16"
  enable_nat_gateway = false
  single_nat_gateway = true

  azs             = ["${var.region}a", "${var.region}b","${var.region}c"]
  public_subnets  = ["192.168.1.0/24", "192.168.2.0/24", "192.168.3.0/24"]

}
```

Creating your own modules (Azure)

- Terraform Documentation on module development
- Important discussion in docs: When to write a module
  - Do not write modules that are just thin wrappers over existing resources
- Very useful: Terraform Standard Module Structure
  - Good in general as a standard for “module structure discipline”
  - Particularly useful if you will use Terraform Cloud/Enterprise – required to import modules into the Terraform Cloud Registry
    - From the doc: The standard module structure is a file and directory layout we recommend for reusable modules distributed in separate repositories. Terraform tooling is built to understand the standard module structure and use that structure to generate documentation, index modules for the module registry, and more.
- Examples:
  - Azure vnet module
  - Non-azure large vmss module

Creating your own modules (AWS)

- Terraform Documentation on module development
- Important discussion in docs: When to write a module
  - Do not write modules that are just thin wrappers over existing resources
- Very useful: Terraform Standard Module Structure
  - Good in general as a standard for “module structure discipline”
  - Particularly useful if you will use Terraform Cloud/Enterprise – required to import modules into the Terraform Cloud Registry
    - From the doc: The standard module structure is a file and directory layout we recommend for reusable modules distributed in separate repositories. Terraform tooling is built to understand the standard module structure and use that structure to generate documentation, index modules for the module registry, and more.
- Note from example in figure that a module is not different to the root modules we have been writing so far in the course – more on this shortly

```
locals {
  server_ami_id = {
    ubuntu = data.aws_ami.ubuntu_22_04.id
    amazon_linux = data.aws_ami.amazon_linux2_kernel_5.id
  }
}

resource "aws_instance" "web_server_instance" {
  ami = local.server_ami_id[var.os]
  instance_type = var.ec2_instance_type

  subnet_id              = var.subnet_id
  vpc_security_group_ids = [var.web_server_sc]

  tags = {
    Name = var.ec2_instance_name
  }
}
```

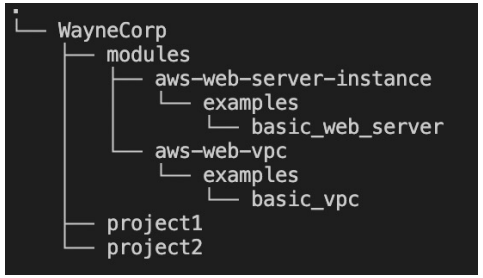
Lab/Demo - Multiple Projects using common modules (AWS)

- From course repo: lab_09b_create
- Contents
  - Simple vpc and web_server modules
  - Modules reused by project1 and project2

```
.
└── WayneCorp
    ├── modules
    │   ├── aws-web-server-instance
    │   │   └── examples
    │   │       └── basic_web_server
    │   └── aws-web-vpc
    │       └── examples
    │           └── basic_vpc
    ├── project1
    └── project2
```

```
WayneCorp/
├── README.md
├── modules
│   ├── aws-web-server-instance
│   │   ├── LICENSE
│   │   ├── README.md
│   │   ├── dependencies.tf
│   │   ├── examples
│   │   │   └── basic_web_server
│   │   │       ├── main.tf
│   │   │       ├── providers.tf
│   │   │       ├── terraform.tfstate
│   │   │       └── variables.tf
│   │   ├── main.tf
│   │   ├── outputs.tf
│   │   └── variables.tf
│   └── aws-web-vpc
│       ├── LICENSE
│       ├── README.md
│       ├── examples
│       │   └── basic_vpc
│       │       ├── main.tf
│       │       ├── outputs.tf
│       │       ├── providers.tf
│       │       ├── terraform.tfstate
│       │       ├── terraform.tfstate.backup
│       │       └── variables.tf
│       ├── locals.tf
│       ├── main.tf
│       ├── outputs.tf
│       ├── providers.tf
│       └── variables.tf
├── project1
│   ├── locals.tf
│   ├── outputs.tf
│   ├── providers.tf
│   ├── root-main.tf
│   ├── terraform.tfstate
│   ├── terraform.tfstate.backup
│   ├── terraform.tfvars
│   └── variables.tf
└── project2
    ├── locals.tf
    ├── outputs.tf
    ├── providers.tf
    ├── root-main.tf
    ├── terraform.tfstate
    ├── terraform.tfstate.backup
    ├── terraform.tfvars
    └── variables.tf
```

Lab/Demo - Multiple Projects using common modules (AWS)



- Suggested tasks:
  - Create ”project3” and use the modules to create a VPC and two instances, one ubuntu and one amazon linux
  - Create a new module, based on aws-web-vpc that creates two public subnets instead of one. Call the module aws-web-vpc-2
    - It could also create a single subnet depending on parameters passed. Hint – similar to official aws vpc
    - Create a new project and use this new vpc module (and aws-web-vpc) to deploy two instances, one in each subnet


```
WayneCorp/
├── README.md
├── modules
│   ├── aws-web-server-instance
│   │   ├── LICENSE
│   │   ├── README.md
│   │   ├── dependencies.tf
│   │   ├── examples
│   │   │   └── basic_web_server
│   │   │       ├── main.tf
│   │   │       ├── providers.tf
│   │   │       ├── terraform.tfstate
│   │   │       └── variables.tf
│   │   ├── main.tf
│   │   ├── outputs.tf
│   │   └── variables.tf
│   └── aws-web-vpc
│       ├── LICENSE
│       ├── README.md
│       ├── examples
│       │   └── basic_vpc
│       │       ├── main.tf
│       │       ├── outputs.tf
│       │       ├── providers.tf
│       │       ├── terraform.tfstate
│       │       ├── terraform.tfstate.backup
│       │       └── variables.tf
│       ├── locals.tf
│       ├── main.tf
│       ├── outputs.tf
│       ├── providers.tf
│       └── variables.tf
├── project1
│   ├── locals.tf
│   ├── outputs.tf
│   ├── providers.tf
│   ├── root-main.tf
│   ├── terraform.tfstate
│   ├── terraform.tfstate.backup
│   ├── terraform.tfvars
│   └── variables.tf
└── project2
    ├── locals.tf
    ├── outputs.tf
    ├── providers.tf
    ├── root-main.tf
    ├── terraform.tfstate
    ├── terraform.tfstate.backup
    ├── terraform.tfvars
    └── variables.tf
```

Module Sources

Module Sources

- Source = where Terraform loads the contents of a module
- Up to now we have seen modules sourced from a local path
  - (e.g. source = "../../modules/web-server") and from the Terraform Registry (official vpc module)
- Many options are available. See TF Module source docs
  - Local paths
  - Terraform Registry (note: includes terraform cloud)
  - GitHub
  - Bitbucket
  - Generic Git, Mercurial repositories
  - HTTP URLs
  - S3 buckets
  - GCS buckets
  - Modules in Package Sub-directories

Module source - github

```
module "static_website" {
  # source = "../../modules/azure-static-website"
  # source = "git@github.com:rpgd60/tf-azure-webstatic.git"
  source = "github.com/rpgd60/tf-azure-webstatic.git?ref=1.0.0"

  # Resource Group
  region  = var.region
  rg_name = "rg-${local.name_postfix}"

  # Storage Account
  storage_account_name              = "${var.environment}staticweb"
  storage_account_tier              = "Standard"
  storage_account_replication_type  = "LRS"
  storage_account_kind              = "StorageV2"
  static_website_index_document     = "index.html"
  static_website_error_404_document = "error.html"
  module_tags                       = local.tags
}
```

- Terraform docs
- Various options for reference, including one for generic git
- Can also refer to different releases using *?ref=x.y.z*
  - Dev could point to 1.1.0 while prod points to 1.0.0
- Multiple Modules under the same repo possible using "double-slash" notation

A special double-slash syntax is interpreted by Terraform to indicate that the remaining path after that point is a sub-directory within the package. For example:

- `hashicorp/consul/aws//modules/consul-cluster`
- `git::https://example.com/network.git//modules/vpc`
- `https://example.com/network-module.zip//modules/vpc`
- `s3::https://s3-eu-west-1.amazonaws.com/examplecorp-terraform-modules/network.zip//modules/vpc`

If the source address has arguments, such as the `ref` argument supported for the version control sources, the sub-directory portion must be *before* those arguments:

- `git::https://example.com/network.git//modules/vpc?ref=v1.2.0`
- `github.com/hashicorp/example//modules/vpc?ref=v1.2.0`

Module source - Azure Repos

- Uses "Generic Git" method to connect to Azure Repos – very similar to Github
- Like Github
  - Can refer to multiple Releases
  - Multiple Modules under the same repo possible using "double-slash" notation
- Lab/Demo:
  - In course repo: /06-tf-modules/04-ado-module
  - Copy files to your Azure Repo prior to this lab
- 2 methods
  - SSH: requires SSH Key pair and installing public key in Azure DevOps
  - HTTPS: requires Azure Devops PAT (personal access token) **hardcoded** in source (BAD IDEA).
    - No var or local allowed. See stackoverflow discussion. See also other stackoverflow discussion. TBC.
    - Note also "/_git/"

```
module "static_website" {
  source = "git@ssh.dev.azure.com:v3/rpgd60/tf-course-01/tf-modules-ado//modules/tf-azure-webstatic-ado"

  # Resource Group
  region  = var.region
  rg_name = "rg-${local.name_suffix}"

  # Storage Account
  storage_account_name             = "${var.environment}staticweb"
  storage_account_tier             = "Standard"
  storage_account_replication_type = "LRS"
  storage_account_kind             = "StorageV2"
  static_website_index_document    = "index.html"
  static_website_error_404_document = "error.html"
  module_tags                      = local.tags
}
```

```
rafa@rp3:dev$ terraform init
Initializing modules...
Downloading git::ssh://git@ssh.dev.azure.com/v3/rpgd60/tf-course-01/tf-modules-ado for static_web
- static_website in .terraform/modules/static_website/modules/tf-azure-webstatic-ado
```

source = git@ssh.dev.azure.com:v3/ORG/PROJECT/REPO//DIR/DIR
source = "git@ssh.dev.azure.com:v3/rpgd60/tf-course-01/tf-modules-ado//modules/tf-password"

source = git::https://<PAT>@dev.azure.com/ORG/PROJECT/_git/REPO//DIR/DIR/"
source = git::https://<PAT>@dev.azure.com/rpgd60/tf-course-01/_git/tf-modules-ado//modules/tf-password/"

Resources and Tips

- Hashicorp Module development docs – Very useful best practices
  - Note that while nested modules are possible, TF recommends in most cases a "flat" structure with a single layer of child modules – they call it "module composition" (example in link)
- Terraform Standard Module Structure
  - Very useful – mentioned also in another slide
- Hashicorp Tutorials - Modules Overview
  - Good intro (generic) and tutorials (AWS specific)

```
$ tree complete-module/
.
├── README.md
├── main.tf
├── variables.tf
├── outputs.tf
├── ...
├── modules/
│   ├── nestedA/
│   │   ├── README.md
│   │   ├── variables.tf
│   │   ├── main.tf
│   │   └── outputs.tf
│   ├── nestedB/
│   └── .../
├── examples/
│   ├── exampleA/
│   │   └── main.tf
│   ├── exampleB/
│   └── .../
```

Terraform Core, Provider and Module versions

- Controlling the versions can help avoid unexpected side-effects ("preventive debugging")
- Best practices from Terraform documentation on version constraints:
  - Module Versions
    - When depending on **third-party** modules, require **specific versions** to ensure that updates only happen when convenient to you.
    - For modules **maintained within your organization**, specifying **version ranges** may be appropriate if semantic versioning is used consistently or if there is a well-defined release process that avoids unwanted updates.
  - Terraform Core and Provider Versions
    - **Reusable modules** should constrain only their minimum allowed versions of Terraform and providers, such as >= 0.12.0. This helps avoid known incompatibilities, while allowing the user of the module flexibility to upgrade to newer versions of Terraform without altering the module.
    - **Root modules** should use a ~> constraint to set both a lower and upper bound on versions for each provider they depend on.
- See also similar recommendations in CloudPosse best practices : use minimum version pinning on all providers
- See also ref to .terraform.lock.hcl file (previous chapters of the course)

```
module "first_vnet" {
  source              = "Azure/vnet/azurerm"
  version             = "2.7.0" # In production use always a
  vnet_name           = "vnet-${local.name_suffix}"
  resource_group_name = azurerm_resource_group.rg.name
```

```
version = ">= 1.2.0, < 2.0.0"
```

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

  required_providers {
    azurerm = {
      #source = "registry.terraform.io/hashicorp/azurerm"
      source  = "hashicorp/azurerm"
      version = "~>3.0"
    }
    aws = {
      source  = "hashicorp/aws"
      version = "~>4.0"
    }
    gcp = {
      source = "hashicorp/google"
      version = "~>4.0"
    }
    random = {
      source  = "hashicorp/random"
      version = "~>3.3"
    }
  }
}
```