Digital Career Institute

AWS SSA

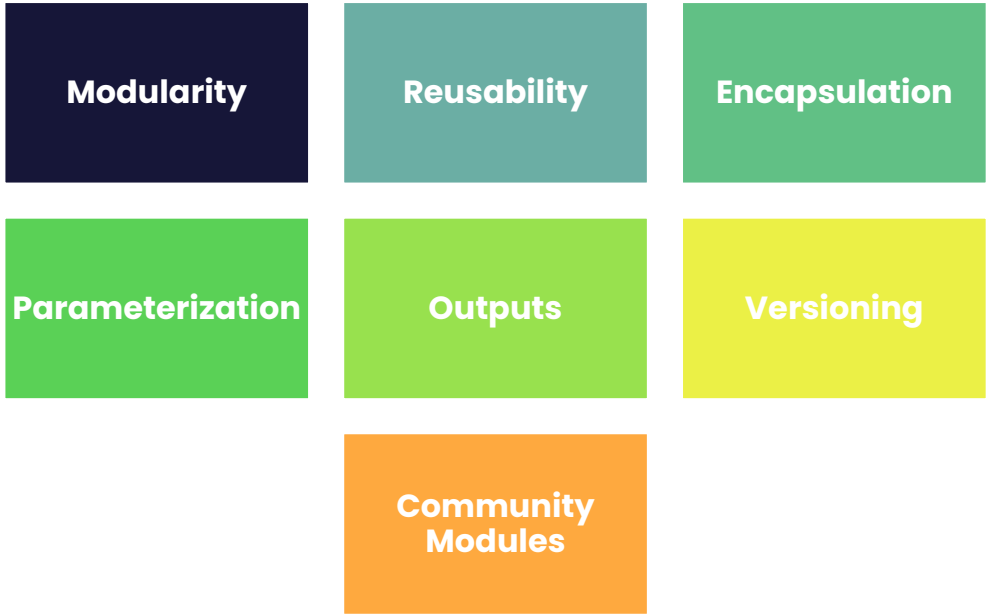
Terraform | Week 1

DAY: 4
Modules

What are Terraform Modules?

- Terraform modules are containers for multiple resources that are used together.
- A module encapsulates Terraform configuration files (.tf files) in a directory.
- Modules enable the reuse of infrastructure configurations.
- They are essentially a collection of .tf files that are managed as a single unit.

Key Aspects of Terraform modules:



Why use modules in Terraform?

- **Reusability:** Write once, use multiple times.
- **Maintainability:** Easier updates and changes.
- **Simplified configurations:** Break down complex configurations into manageable parts.
- **Team collaboration:** Promotes collaboration by enabling different teams to use the same infrastructure code.

Overview of modularity in Terraform configurations

- **Modularity:** The practice of splitting configurations into smaller, self-contained components.
- **Root Module:** The main working directory where your Terraform configuration starts.
- **Child Modules:** Reusable units that the root module calls to manage specific parts of the infrastructure.
- **Module Composition:** Building more complex infrastructures by combining simple modules.

Input variables in Terraform

- Allow modules to accept values at runtime to customize behavior.
- Defined using the variable block.

```
variable "instance_type" {
  type    = string
  default = "t2.micro"
}
```

- Customize infrastructure without modifying the module.

Output values

- Expose values from a module for use by other modules or configurations.
- Defined using the output block.

```
output "vpc_id" {
  value = aws_vpc.main.id
}
```

-
- Outputs make values accessible for downstream use.

Module Usage and Configuration in HCL

Output Variables:

Modules can also define output variables using the output block. These allow you to retrieve values from the module.

```hcl
output "example_output" {
  description = "An example output variable"
  value       = some_resource.some_attribute
}
```

Calling Module Outputs:

In your main configuration, you can reference the output variables from the module like this:

```hcl
resource "some_resource" "example" {
  attribute = module.example_module.example_output
}
```

Module Usage and Configuration in HCL

Using Module Outputs:

The output from the module can be used just like any other variable in your configuration.

Module Configuration Version Constraints:

It's a good practice to specify version constraints for your modules in the versions.tf file to ensure that the correct version of the module is used.

Running Terraform Commands:

When you run Terraform commands (e.g., terraform init, terraform apply), Terraform will automatically download and manage the module dependencies.

```hcl
terraform {
  required_providers {
    aws = {
      source  = "hashicorp/aws"
      version = ">= 3.0, < 4.0"
    }
  }
}
```

Local values

- Store intermediate values for use within the module.
- Defined using the locals block.

```
locals {
  instance_name = "${var.env}-app-server"
}
```

- Help simplify and reuse logic within the module.

Advantages of Using Modules

Reusability	Write once, use many times.
Maintainability	Easier updates and scaling.
Simplified configuration	Break down complexity into manageable parts.
Collaboration and team scaling	Promote standardization across teams.

Basic folder structure of a module

- A typical module directory structure might look like this:

```bash
module_directory/
├── main.tf        # Contains resource definitions
├── variables.tf   # Contains variable definitions
├── outputs.tf     # Contains output variable definitions
└── README.md      # Documentation
```

- By following these steps, you can create, use, and configure modules in HCL to modularize and organize your infrastructure code effectively, making it easier to manage and maintain complex configurations.

Example: Basic Module Structure

- **Module Folder Structure**:
  - `my-ec2-module/`
    - `main.tf`
    - `variables.tf`
    - `outputs.tf`
    - `README.md`

```
my-ec2-module/
├── main.tf
├── variables.tf
├── outputs.tf
└── README.md
```

Example: main.tf – Defining Resources

Example `main.tf` for creating an EC2 instance:

```
resource "aws_instance" "example" {
  ami           = var.ami
  instance_type = var.instance_type

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

Example: variables.tf — Defining Input Variables

Example `variables.tf` for customizing the module:

```
variable "ami" {
  type    = string
  default = "ami-12345678"
  description = "The ID of the AMI to use for the instance."
}

variable "instance_type" {
  type    = string
  default = "t2.micro"
  description = "The type of EC2 instance to launch."
}

variable "instance_name" {
  type    = string
  default = "example-instance"
  description = "The name to assign to the EC2 instance."
}
```

Example: outputs.tf — Defining Output Values

Example `outputs.tf` for exposing key values:

```
output "instance_id" {
  value       = aws_instance.example.id
  description = "The ID of the EC2 instance."
}

output "instance_public_ip" {
  value       = aws_instance.example.public_ip
  description = "The public IP address of the EC2 instance."
}
```

Example: Using the Module

Example of how to call the module:

```
module "ec2_instance" {
  source = "./my-ec2-module"

  ami           = "ami-87654321"
  instance_type = "t3.micro"
  instance_name = "my-app-server"
}
```

Example of Input Variables

String Variable Example:

```
variable "ami" {
  type    = string
  default = "ami-12345678"
  description = "The ID of the AMI to use for the EC2
instance."
}
```

Boolean Variable Example:

```
variable "enable_monitoring" {
  type    = bool
  default = true
  description = "Enable or disable EC2 monitoring."
}
```

List and Map Input Variables

List Variable Example:

```
variable "availability_zones" {
  type    = list(string)
  default = ["us-east-1a", "us-east-1b"]
  description = "List of availability zones for the subnets."
}
```

Map Variable Example:

```
variable "tags" {
  type    = map(string)
  default = {
    Name        = "example-instance"
    Environment = "dev"
  }
  description = "Tags to apply to the EC2 instance."
}
```

Default Values and Type Definitions

Default Values:

- Provide sensible defaults.
- Users can override these defaults if needed.

Type Definitions:

- Always define types for clarity.
- Supported types: `string`, `number`, `bool`, `list`, `map`, etc.

Validating Input Variables

Validation Blocks: Enforce rules for input variables.

```
variable "instance_type" {
  type = string
  description = "The type of EC2 instance."

  validation {
    condition     = contains(["t2.micro", "t3.micro"],
var.instance_type)
    error_message = "Instance type must be t2.micro or
t3.micro."
  }
}
```

Example: Passing Input Variables

Using Input Variables in a Configuration:

```
module "ec2_instance" {
  source = "./my-ec2-module"
  ami           = "ami-87654321"
  instance_type = "t3.micro"
  enable_monitoring = true
}
```

Example Output Values

Basic Output Example:

```
output "instance_id" {
  value       = aws_instance.example.id
  description = "The ID of the EC2 instance."
}
```

Exposing IP Address:

```
output "instance_public_ip" {
  value       = aws_instance.example.public_ip
  description = "The public IP address of the EC2 instance."
}
```

Using Outputs in Other Modules

Referencing Output Values in a Parent Module:

```
module "ec2_instance" {
  source = "./my-ec2-module"
}

output "instance_id_from_module" {
  value = module.ec2_instance.instance_id
}
```

Defining Multiple Output Values

Outputting Multiple Values:

```
output "subnet_ids" {
  value       = aws_subnet.public.*.id
  description = "List of IDs for the public subnets."
}

output "vpc_id" {
  value       = aws_vpc.main.id
  description = "The ID of the VPC."
}
```

Conclusion: Defining and Using Output Values

- Use **output values** to expose important resource attributes.
- **Provide clear names** and **descriptions** for outputs.
- Use **sensitive outputs** for confidential information.
- Limit outputs to avoid clutter and unnecessary exposure.

Module Source Types – Local path

- **Local Path**: Modules stored locally on your machine.
- **Use Case**: Quick development and testing of modules.

```
module "ec2_instance" {
  source = "./modules/ec2-instance"
  instance_type = "t2.micro"
  ami = "ami-12345678"
}
```

Benefits of Using Local Path Modules

- **Rapid Development**: Ideal for quickly iterating on module code during development.
- **No External Dependencies**: No need for internet access or version control during development.
- **Easy to Share**: Can be shared within a team via a local directory structure.

When to Use Local Path Modules

- **Module Development**: Perfect for building new modules and testing locally.
- **Small Projects**: Suitable for small-scale projects where teams are co-located.
- **Testing and Prototyping**: Quickly test new features or infrastructure designs.

Benefits of Using Remote Repository Modules

- **Version Control**: Track changes and revert to previous versions as needed.
- **Collaboration**: Easily share and collaborate on infrastructure modules across teams.
- **Consistency**: Ensure all environments use the same infrastructure code.
- **Scalability**: Centralized management of modules for multiple projects.

Module Source Types – Remote repositories

- **Remote Repositories**: Source modules from remote version control systems.
- **Common Providers**: GitHub, GitLab, Bitbucket.
- **Use Case**: Share and version modules across teams and projects.

```
module "vpc" {
  source = "git::https://github.com/user/repo.git//modules/vpc?ref=v1.0.0"
  cidr   = "10.0.0.0/16"
}
```

Module Source Types – Terraform Registry

- **Terraform Registry**: Official source for reusable, versioned modules.
- **Use Case**: Access well-maintained, community, and provider-supported modules.

```
module "vpc" {
  source  = "terraform-aws-modules/vpc/aws"
  version = "3.0.0"
  cidr    = "10.0.0.0/16"
}
```

Benefits of Using the Terraform Registry

- **Best Practices**: Use community-vetted, well-maintained modules.
- **Simplicity**: Avoid reinventing the wheel by using pre-built modules.
- **Security**: Many modules are reviewed and maintained by trusted sources.
- **Time-Saving**: Speeds up deployment by providing ready-to-use modules.

Best Practices for Module Design

- Write small, reusable modules
- Avoid hard-coding values
- Use versioned modules
- Separate infrastructure concerns

What are remote modules?

Remote Modules: Modules sourced from outside your local directory.
Can be stored in:

- Terraform Registry
- Version control repositories (GitHub, Bitbucket, etc.)
- Object storage (e.g., S3, GCS)

Use `source` to reference the module's location.

```
module "vpc" {
  source = "terraform-aws-modules/vpc/aws"
  version = "2.0.0"
}
```

Accessing modules from Terraform Registry

Terraform Registry: Official source for reusable Terraform modules.
Provides public and private modules for various cloud platforms.
Use the `source` argument to access modules.

```
module "vpc" {
  source  = "terraform-aws-modules/vpc/aws"
  version = "3.0.0"
  cidr    = "10.0.0.0/16"
}
```

Example of using a remote module

Using a remote VPC module from the Terraform Registry:

```
module "vpc" {
  source  = "terraform-aws-modules/vpc/aws"
  version = "3.0.0"
  cidr    = "10.0.0.0/16"
  azs     = ["us-east-1a", "us-east-1b", "us-east-1c"]
  public_subnets  = ["10.0.1.0/24", "10.0.2.0/24", "10.0.3.0/24"]
  private_subnets = ["10.0.101.0/24", "10.0.102.0/24", "10.0.103.0/24"]
  enable_nat_gateway = true
}
```

Best practices for managing module outputs

- Limit Exposure of Outputs
- Use Descriptive Names
- Avoid Sensitive Data in Outputs
- Optimize Outputs for Reusability
- Version Control of Outputs

What are nested modules?

Nested Modules: Modules within modules, allowing more complex infrastructures.
Organizes infrastructure into logical layers or components.
Helps break down complex configurations into manageable, reusable pieces.

```
module "networking" {
  source = "./modules/network"
}

module "compute" {
  source = "./modules/compute"
  vpc_id = module.networking.vpc_id
}
```

Organizing Infrastructure with Nested Modules

Why Use Nested Modules?

- Modularizes infrastructure at different levels (e.g., networking, compute).
- Makes the configuration more readable and easier to maintain.
- Promotes reusability of each module component.

Benefits of Nested Modules

- **Scalability**:
  Manage complex infrastructures by layering responsibilities.
- **Reusability**:
  Reuse child modules in different parts of the infrastructure.
- **Modularity**:
  Break down configurations into smaller, easier-to-maintain components.

Best Practices for Nested Modules

- **Maintain a Clear Hierarchy:**
  Keep modules logically organized with a clear parent-child relationship.
- **Limit Dependencies:**
  Avoid creating too many interdependent modules to reduce complexity.
- **Test Each Module Separately:**
  Ensure that each module works on its own before nesting.

Documenting input variables and outputs

- **Input Variables**: Provide clear descriptions and defaults.
- **Outputs**: Describe purpose and structure.
- Use description argument for clarity.
- Example of input variable documentation:

Common Module Pitfalls

- Overcomplicating Modules
- Hardcoding Values
- Lack of Version Control
- Ignoring Module Outputs
- Insufficient Documentation

Overcomplicating Modules

- **Problem**: Too many resources or responsibilities in one module.
- **Consequence**: Harder to maintain, troubleshoot, and reuse.
- **Solution**: Break down modules into smaller, focused components.

Hardcoding Values

- **Problem**: Hardcoded values reduce flexibility.
- **Consequence**: Modules become difficult to reuse in different environments.
- **Solution**: Use input variables instead of hardcoding.

Lack of Version Control

- **Problem**: Not pinning module versions.
- **Consequence**: Risk of breaking infrastructure due to upstream changes.
- **Solution**: Use version control by specifying the `version` attribute.

Ignoring Module Outputs

- **Problem**: Not using or defining useful outputs.
- **Consequence**: Key resource data becomes difficult to access.
- **Solution**: Define and utilize outputs for resource IDs, IPs, etc.

Insufficient Documentation

- **Problem**: Lack of clear descriptions and usage instructions.
- **Consequence**: Modules are harder to use and maintain.
- **Solution**: Provide detailed documentation and comments.

Avoiding Common Pitfalls

- Keep modules simple and focused.
- Avoid hardcoding values—use input variables.
- Pin module versions for consistency.
- Define useful outputs for resource attributes.
- Provide detailed documentation and follow best practices.

Digital Career Institute

Next up

Terraform | Week 2

DAY 1:
Modules: Design and Usage

LEARN FOR **A NEW LIFE.**

DCI