What are Terraform Modules?

A Terraform module is a container for multiple resources that are used together. It is the building block of Terraform configurations and promotes code reusability, modularity, and consistency.

Advantages of Terraform Modules

Advantage	Explanation
1. Reusability	Write once, reuse multiple times across environments (dev, prod, staging).
2. Standardization	Ensures consistency (naming, tagging, security rules) across the organization.
3. Scalability	Easier to manage large and complex infrastructures by breaking them into small pieces.
4. Code Organization	Cleaner, modular code — makes big Terraform projects manageable and readable.
5. Easier Maintenance	Fixing a bug or updating logic in one place automatically updates everywhere it's used.
6. Collaboration Friendly	Different teams can work on different modules independently (network, compute, etc.).
7. Testing becomes easier	You can unit-test a module separately before integrating.
8. Faster Development	Build new projects faster by combining existing modules rather than writing new code.

❌Disadvantages of Terraform Modules

Disadvantage	Explanation

Disadvantage	Explanation
1. Learning Curve	Beginners may find module structure (input variables, outputs) confusing.
2. Debugging is harder	When something fails, finding where (main or module) the error happened can take time.
3. Overhead for small projects	For very small projects, creating modules may feel like over-engineering.
4. Version Management	When you update a shared module, you must be careful to not break dependent projects.
5. Increased Complexity	Too many nested modules (modules calling modules) can make the project hard to understand.
6. Refactoring pain	If you make major changes in module inputs/outputs, dependent code needs updates.
7. Dependency Management	Sometimes module dependencies aren't clear and cause planning/apply issues if not handled properly.

🎯Quick Real-Life Example

- **Without modules**:
    - You copy-paste EC2 code 10 times for 10 projects.
    - Later, when you need to add a tag to EC2, you must edit 10 files.
- **With modules**:
    - EC2 code is inside one ec2-instance-module.
    - All 10 projects just call the module.
    - You add a tag once in the module → automatically available everywhere!

✨When Should You Use Modules?

✅ Use modules when:

- Infrastructure is growing.
- You have multiple environments (dev, prod, QA).
- Teams want consistent infrastructure design.
- You want to reuse code across projects.

❌ Avoid modules when:

- The project is extremely small (e.g., just 1 VM and 1 security group).

⚡ **In Simple Words**

"Modules in Terraform are like **functions in programming** —
instead of copying-pasting code, you write once and reuse smartly."

Sample Terraform Module Folder Structure

```
css

CopyEdit

project-root/

│

├── main.tf

├── variables.tf

├── outputs.tf

├── terraform.tfvars (optional)

├── provider.tf

│

├── modules/

│   ├── vpc/

│   │   ├── main.tf

│   │   ├── variables.tf
```

```
│   │   ├── outputs.tf
│   │
│   ├── ec2/
│   │   ├── main.tf
│   │   ├── variables.tf
│   │   ├── outputs.tf
│   │
│   ├── security-group/
│   │   ├── main.tf
│   │   ├── variables.tf
│   │   ├── outputs.tf
│   │
│   └── s3/
│       ├── main.tf
│       ├── variables.tf
│       ├── outputs.tf
│
├── environments/
│   ├── dev/
│   │   ├── main.tf
│   │   ├── variables.tf
│   │   ├── terraform.tfvars
│   │
│   ├── prod/
```

```
│   │   ├── main.tf
│   │   ├── variables.tf
│   │   ├── terraform.tfvars
│   │
│   └── README.md
```

📚**Explanation:**

Folder/File	Purpose
main.tf	Root Terraform file where you call modules like module "vpc" {}.
variables.tf	Declares input variables for the root project.
outputs.tf	Declares output values (maybe IPs, IDs) from root.
provider.tf	Defines AWS, Azure, GCP providers and their versions.
terraform.tfvars	Default values for variables (e.g., region = "us-east-1").
modules/	Directory containing reusable modules like VPC, EC2, SG, S3, etc.
modules/<module>	Each module has its own isolated Terraform files (main/variables/outputs).
environments/	Separate environment-specific configurations (dev, prod, QA, etc.).
README.md	Documentation explaining project and module usage.

🔥**Inside the main.tf at project root**

hcl

CopyEdit

```
module "vpc" {
  source = "./modules/vpc"
```

```
  vpc_cidr = "10.0.0.0/16"
  region   = var.region
}

module "ec2" {
  source = "./modules/ec2"

  instance_type = "t2.micro"
  ami_id        = var.ami_id
  subnet_id     = module.vpc.public_subnet_id
}
```

🎯**Good Practice Tips:**

- Always create **main.tf**, **variables.tf**, and **outputs.tf** inside each module.
- Name module folders **short and clear** (ec2, vpc, alb, rds).
- Keep **modules generic** (pass values through variables, don’t hardcode).
- Use **version pinning** for providers in provider.tf.
- Keep a **README.md** inside each module to explain its usage if your project is big.

Common Terraform Module Sources

Source Type	Explanation & Examples
Local File Path	Use a module stored locally in your project directory.
Terraform Registry	Use public or private modules hosted on the Terraform Registry.
GitHub Repository	Use a module stored in a GitHub repository.
Git URL (any Git service)	Use a module stored in any Git-based repository (e.g., GitLab, Bitbucket).

Source Type	Explanation & Examples
S3 Bucket	Use a module stored in an S3 bucket.
HTTP URL	Use a module stored at a URL that returns a .zip file (e.g., HTTP or HTTPS URL).

1. Local File Path

You can use a module stored locally in your project directory, typically within the modules/ folder.

Example:

hcl

CopyEdit

```
module "vpc" {
  source = "./modules/vpc"  # Local directory path
  region = "us-west-2"
}
```

2. Terraform Registry

The Terraform Registry is a collection of publicly available modules created by the community and HashiCorp. You can use these modules without needing to write your own from scratch.

Example:

hcl

CopyEdit

```
module "vpc" {
  source = "terraform-aws-modules/vpc/aws"  # Registry module
  version = "~> 3.0"  # Optional: specify version
  cidr = "10.0.0.0/16"
  enable_dns_support = true
```

```
  enable_dns_hostnames = true
}
```

You can find modules in the Terraform Registry by browsing the Terraform AWS Modules or other modules created for specific providers or use cases.

3. GitHub Repository

You can specify a GitHub repository as the source. Terraform supports fetching modules directly from GitHub repositories, whether they are public or private.

Example:

hcl

CopyEdit

```
module "vpc" {
  source = "github.com/username/repository"  # GitHub URL
  version = "v1.0.0"
  cidr = "10.0.0.0/16"
}
```

If you want to specify a particular branch, tag, or commit hash:

hcl

CopyEdit

```
module "vpc" {
  source = "github.com/username/repository//subfolder?ref=v1.0.0"
  cidr = "10.0.0.0/16"
}
```

4. Git URL (Any Git-based Repository)

You can also use Git repositories hosted on other platforms like GitLab or Bitbucket.

Example (GitLab):

hcl

CopyEdit

```
module "vpc" {
  source = "git::https://gitlab.com/username/repository.git//modules/vpc"
  region = "us-west-2"
}
```

This can also work with private Git repositories by specifying the correct authentication (e.g., using SSH keys or token-based access).

5. S3 Bucket

Modules can also be retrieved from an **S3 bucket**, which is useful for managing private modules or modules that you want to store externally.

Example:

hcl

CopyEdit

```
module "vpc" {
  source = "s3::https://s3.amazonaws.com/mybucket/my-module.zip"
}
```

You would need to upload the .zip file of the module to your S3 bucket before referencing it.

6. HTTP URL

If you have a module stored as a .zip file on an HTTP or HTTPS server, you can use it directly.

Example:

```
module "vpc" {
  source = "http://example.com/modules/my-module.zip"
}
```

This is especially useful when sharing modules between teams or storing them in centralized locations.

🧰Key Terraform Module Source Examples

Module Source	Example	Use Case
Local File Path	source = "./modules/vpc"	Use for custom local modules.
Terraform Registry	source = "terraform-aws-modules/vpc/aws"	Use widely used, pre-built modules.
GitHub Repository	source = "github.com/user/repository"	Private or public custom modules.
Git URL	source = "git::https://gitlab.com/user/repo.git"	Private/custom modules in Git.
S3 Bucket	source = "s3::https://s3.amazonaws.com/mybucket/my-module.zip"	Private modules stored in S3.
HTTP URL	source = "http://example.com/modules/my-module.zip"	Hosting modules on a web server.

⚡ Summary of Terraform Module Sources:

- **Local file path**: For locally-developed, project-specific modules.
- **Terraform Registry**: A vast collection of pre-made modules.
- **GitHub/GitLab/Bitbucket**: Fetch modules from repositories for custom, version-controlled code.
- **S3 Bucket**: Store and retrieve private modules via AWS S3.
- **HTTP URL**: Fetch .zip-packed modules hosted on a web server.

step-by-step guide to create a reusable **Terraform AWS VPC module from scratch** and then **consume it** from another module (or root module):

✅ Step 1: Create folder structure

```
css

CopyEdit

my-terraform-project/
├── main.tf
├── variables.tf
├── outputs.tf
├── providers.tf
└── modules/
    └── vpc/
        ├── main.tf
        ├── variables.tf
        ├── outputs.tf
```

✅ Step 2: Write code for the VPC module

📁 modules/vpc/main.tf

```
hcl

CopyEdit

resource "aws_vpc" "main" {
  cidr_block           = var.vpc_cidr
  enable_dns_support   = true
  enable_dns_hostnames = true
```

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

📁modules/vpc/variables.tf

h

CopyEdit

```
variable "vpc_cidr" {
  type        = string
  description = "CIDR block for the VPC"
}

variable "vpc_name" {
  type        = string
  description = "Name tag for the VPC"
}
```

📁modules/vpc/outputs.tf

hcl

CopyEdit

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

✅ **Step 3: Use the VPC module in root/main configuration**

📁main.tf

hcl

CopyEdit

```
module "vpc" {
  source    = "./modules/vpc"
  vpc_cidr  = "10.0.0.0/16"
  vpc_name  = "my-main-vpc"
}
```

📁 providers.tf

hcl

CopyEdit

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

📁 outputs.tf

hcl

CopyEdit

```
output "vpc_id" {
  value = module.vpc.vpc_id
}
```

✅ Step 4: Run Terraform commands

In your terminal:

bash

CopyEdit

```
cd my-terraform-project
```

```
terraform init        # Initialize the project and download providers/modules
terraform plan        # Preview the infrastructure changes
terraform apply       # Apply the changes and create the VPC
```

📦**Summary:**

- ✅ **Reusable module** is in modules/vpc
- ✅ **Root module** calls it via module "vpc" { source = "./modules/vpc" }
- ✅ You separate logic from usage — a Terraform best practice.

Terraform module best practices that help ensure **reusability**, **scalability**, and **maintainability** of your infrastructure code:

✅ 1. Use Modules to Reuse Code

- Create modules for repeatable components like VPCs, subnets, EC2 instances, security groups, etc.
- Avoid hardcoding values; pass variables from root modules.

✅ 2. Follow a Clear Directory Structure

css

CopyEdit

```
project/
├── main.tf
├── variables.tf
├── outputs.tf
└── modules/
    └── vpc/
```

├── main.tf

├── variables.tf

├── outputs.tf

✅ 3. Use Meaningful Variable Names and Descriptions

- Every input/output should be well-described in variables.tf and outputs.tf.

hcl

CopyEdit

```
variable "vpc_cidr" {
  description = "CIDR block for the VPC"
  type        = string
}
```

✅ 4. Avoid Using Default Values for Critical Variables

- Force users to pass values to avoid accidental configurations.

✅ 5. Pin Provider Versions

hcl

CopyEdit

```
terraform {
  required_providers {
    aws = {
      source  = "hashicorp/aws"
      version = "~> 5.0"
    }
  }
```

```
}
```

✅ 6. Use Versioned Modules

- Use version = "x.y.z" if pulling from the Terraform Registry or a Git source.
- Tag releases in Git (v1.0.0) for better control and rollback.

✅ 7. Use locals to Simplify Logic Inside Modules

hcl

CopyEdit

```
locals {
  tags = {
    Environment = var.environment
    Project     = var.project_name
  }
}
```

✅ 8. Expose Only Required Outputs

- Keep modules clean by exposing only needed information to the root module.

✅ 9. Avoid Resource Duplication

- Use count or for_each wisely to control the number of resources created.

hcl

CopyEdit

```
resource "aws_subnet" "this" {
  for_each = var.subnet_configs
  ...
```

}

✅ 10. Document Your Module

- Add a README.md with:
    - Purpose
    - Inputs/Outputs
    - Examples
    - Required providers

✅ 11. Use terraform fmt and terraform validate

- Keep your code clean and validated.

✅ 12. Test Modules Before Use

- Use separate test environments or examples/ folder to validate behavior.

✅ 13. Use Remote State for Module Outputs

- When modules are split across projects, use terraform_remote_state to fetch outputs from other modules' state.

✅ 14. Separate Production and Development Configs

- Use workspaces or separate backends for dev, staging, and prod.

✅ 15. Use .terraformignore

- Exclude unnecessary files from being uploaded to Terraform Cloud/Enterprise.

Using one Terraform module's **output as input to another module** is a common and powerful practice, especially when dealing with modular, layered infrastructure (e.g., using a VPC ID from a vpc module in an ec2 module).

✅ Use Case Example: VPC and EC2 Modules

🎯 Scenario:

You have:

- A vpc module that creates a VPC and outputs its ID.
- An ec2 module that needs the VPC ID to create a security group or launch an EC2 instance in that VPC.

📁 Folder Structure

```
css

CopyEdit

terraform-project/

├── main.tf

├── modules/

│   ├── vpc/

│   │   └── main.tf, variables.tf, outputs.tf

│   └── ec2/

│       └── main.tf, variables.tf
```

🧱 1. modules/vpc/outputs.tf

```
h

CopyEdit

output "vpc_id" {

  value = aws_vpc.main.id
```

```
}
```

🧱2. modules/ec2/variables.tf

```
hcl

CopyEdit

variable "vpc_id" {

  type        = string

  description = "VPC ID where EC2 will be created"

}
```

🧱3. Root main.tf (Wiring it all together)

```
hcl

CopyEdit

module "vpc" {

  source    = "./modules/vpc"

  vpc_cidr  = "10.0.0.0/16"

  vpc_name  = "my-vpc"

}


module "ec2" {

  source   = "./modules/ec2"

  vpc_id   = module.vpc.vpc_id  # <– using output from vpc module

  ami_id   = "ami-0abcd1234efgh5678"

  instance_type = "t2.micro"

}
```

🧱4. modules/ec2/main.tf (Using the input)

h

CopyEdit

```
resource "aws_security_group" "allow_ssh" {
  name        = "allow_ssh"
  description = "Allow SSH"
  vpc_id      = var.vpc_id  # <-- input from root module
  ingress {
    from_port   = 22
    to_port     = 22
    protocol    = "tcp"
    cidr_blocks = ["0.0.0.0/0"]
  }
}
```

✅Other Real-world Examples

Module A Output	Module B Input Usage
subnet_id	Launch EC2 instance inside the subnet
alb_dns_name	Output DNS for another module to add Route53 record
s3_bucket_name	Used in a lambda module to configure logging
rds_instance_endpoint	Used in app module for DB connection

Goal:

Provision the following using Terraform modules:

- **VPC Module** → creates VPC
- **Subnet Module** → creates subnet using VPC ID from VPC module
- **EC2 Module** → launches EC2 instance in the subnet

📁**Folder Structure:**

```
terraform-project/
├── main.tf
├── providers.tf
├── variables.tf
├── outputs.tf
├── modules/
│   ├── vpc/
│   │   ├── main.tf
│   │   ├── variables.tf
│   │   └── outputs.tf
│   ├── subnet/
│   │   ├── main.tf
│   │   ├── variables.tf
│   │   └── outputs.tf
│   └── ec2/
│       ├── main.tf
│       ├── variables.tf
│       └── outputs.tf
```

🔹 **providers.tf**

hcl

CopyEdit

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

- **main.tf (Root)**

h

CopyEdit

```
module "vpc" {
  source    = "./modules/vpc"
  vpc_cidr  = "10.0.0.0/16"
  vpc_name  = "my-vpc"
}

module "subnet" {
  source   = "./modules/subnet"
  subnet_cidr = "10.0.1.0/24"
  vpc_id   = module.vpc.vpc_id
  subnet_name = "public-subnet"
}

module "ec2" {
  source   = "./modules/ec2"
  subnet_id = module.subnet.subnet_id
  ami_id    = "ami-0c55b159cbfafe1f0"
```

```
  instance_type = "t2.micro"
}
```

✅ **modules/vpc**

variables.tf

```
hcl
CopyEdit
variable "vpc_cidr" {}
variable "vpc_name" {}
```

main.tf

```
resource "aws_vpc" "main" {
  cidr_block = var.vpc_cidr
  tags = {
    Name = var.vpc_name
  }
}
```

outputs.tf

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

✅ **modules/subnet**

variables.tf

```
hcl
```

CopyEdit

```
variable "vpc_id" {}

variable "subnet_cidr" {}

variable "subnet_name" {}
```

main.tf

```
resource "aws_subnet" "main" {
  vpc_id     = var.vpc_id
  cidr_block = var.subnet_cidr
  tags = {
    Name = var.subnet_name
  }
}
```

outputs.tf

hcl

CopyEdit

```
output "subnet_id" {
  value = aws_subnet.main.id
}
```

✅ modules/ec2

variables.tf

hcl

CopyEdit

```
variable "subnet_id" {}

variable "ami_id" {}

variable "instance_type" {}
```

main.tf

hcl

CopyEdit

```
resource "aws_instance" "web" {
  ami           = var.ami_id
  instance_type = var.instance_type
  subnet_id     = var.subnet_id
  tags = {
    Name = "WebServer"
  }
}
```

🧪**Deployment Steps:**

bash

CopyEdit

```
cd terraform-project/
terraform init
terraform plan
terraform apply
```

✅**Outputs and Flow Summary:**

- module.vpc → creates VPC and outputs vpc_id
- module.subnet → takes vpc_id, creates subnet, outputs subnet_id
- module.ec2 → uses subnet_id, creates EC2