DevOps Shack

Modular Mastery

BUILDING REUSABLE TERRAFORM INFRASTRUCTURTE

By DevOps Shack

[Click here for DevSecOps & Cloud DevOps Course]

DevOps Shack

Modular Mastery:

Building Reusable Terraform Infrastructure

Table of Contents

1. Understanding Terraform Modules

1.1 What Are Terraform Modules?

A **Terraform module** is a container for multiple resources that are used together. Modules allow you to group related resources (e.g., networking, compute, storage) and reuse them across different configurations, reducing duplication and enhancing maintainability.

Think of a module as a **function in programming**: it takes input variables, performs some operations (defines resources), and returns outputs.

There are **three types of modules** in Terraform:

- **Root Module** – the primary working directory with the main Terraform configuration (main.tf, variables.tf, etc.)
- **Local Modules** – modules located within the same repository, used with relative paths.
- **Remote Modules** – modules fetched from external sources like Terraform Registry or Git repositories.

1.2 Benefits of Using Modules

Modules bring several advantages to Terraform-based infrastructure:

Benefit	Description
Reusability	Define once, use in multiple places or projects.
Consistency	Apply standard configurations across environments.
Maintainability	Isolate and update logic in a single module, reducing drift.
Scalability	Manage large infrastructures by breaking them into logical parts.
Separation of Concerns	Keep code organized by domain (network, compute, storage).

1.3 Anatomy of a Terraform Module

A basic module typically contains:

File	**Purpose**
main.tf	The core logic defining resources.
variables.tf	Input variable definitions with types and descriptions.
outputs.tf	Outputs returned after module execution.
providers.tf *(optional)*	Provider configuration (often inherited from the root module).
README.md *(optional)*	Documentation for usage, inputs, and outputs.

Example Directory Structure:

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

1.4 When to Use Modules

Use modules when:

- You repeat similar resource blocks across projects.
- Managing infrastructure across multiple environments (dev/staging/prod).
- You want to create a standard library of infrastructure components for your team or organization.

Don't over-engineer by modularizing every small resource—start modularizing when:

- The logic is reused 2–3 times or more.

- It involves a complex set of tightly related resources (e.g., setting up a VPC).

1.5 Summary

Terraform modules are the foundation of reusable, maintainable, and scalable infrastructure code. By understanding their structure and purpose, you set the stage for efficient infrastructure management.

2. Setting Up and Structuring Terraform Projects

2.1 Installing Terraform and Tooling

Before writing any Terraform code, ensure your development environment is properly set up.

Install Terraform

You can install Terraform via Terraform's official site or use a version manager like tfenv to manage multiple versions:

```
# Install tfenv (macOS/Linux)

brew install tfenv


# Install a specific Terraform version

tfenv install 1.6.2


# Set it as the default

tfenv use 1.6.2
```

2.2 Recommended Directory Structure

A clear, modular directory layout ensures scalability and collaboration.

Basic Structure (Monolith Project)

```
project-root/
├── main.tf
├── variables.tf
├── outputs.tf
├── terraform.tfvars
├── modules/
│   └── networking/
│       ├── main.tf
```

```
|       ├── variables.tf
|       └── outputs.tf
└── environments/
    ├── dev/
    |   └── main.tf
    └── prod/
        └── main.tf
```

Root Module vs Child Modules

- **Root Module:** where terraform init, plan, and apply are run.
- **Child Modules:** used by the root, typically stored in modules/ or pulled remotely.

2.3 Organizing for Environment-Specific Deployments

It's common to separate infrastructure by environments:

```
environments/
├── dev/
|   └── main.tf
├── staging/
|   └── main.tf
└── prod/
    └── main.tf
```

Each can reuse the same modules with different variable values (terraform.tfvars), ensuring consistency and isolation.

2.4 Terraform Backends (Optional but Recommended)

Use remote backends (like AWS S3, Terraform Cloud, or Azure Blob) to store Terraform state safely and allow collaboration.

Example (S3 backend):

```
terraform {
  backend "s3" {
    bucket         = "my-terraform-states"
    key            = "networking/dev/terraform.tfstate"
    region         = "us-west-2"
    dynamodb_table = "terraform-locks"
    encrypt        = true
  }
}
```

2.5 Provider Configuration

Define providers in the root module to ensure all child modules inherit it:

```
provider "aws" {
  region = var.region
}
```

Avoid configuring providers inside modules unless you want to support **multi-provider** modules (advanced use case).

2.6 Formatting and Style

Keep your code readable and consistent:

- Use terraform fmt to format code.
- Use comments and consistent naming conventions.
- Create a README.md for each module with usage instructions.

2.7 Example Usage with Local Module

In your root main.tf, reference a local module:

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

  vpc_name = "dev-vpc"
  cidr_block = "10.0.0.0/16"
  region = "us-west-2"
}
```

2.8 Summary

A well-structured Terraform project is essential for team collaboration and long-term maintainability. By separating environments, using remote state, and organizing your modules, you set a solid foundation for scalable infrastructure.

3. Writing Reusable and Scalable Modules

3.1 Inputs, Outputs, and Variables

Terraform modules use input variables to receive values and output blocks to return results.

Input example (variables.tf):

```
variable "vpc_cidr" {
  description = "CIDR block for the VPC"
  type        = string
  default     = "10.0.0.0/16"
}
```

Output example (outputs.tf):

hcl

CopyEdit

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

In the module consumer:

hcl

CopyEdit

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

3.2 Input Validation with validation Blocks

Enhance input safety using built-in validation:

```
variable "environment" {
  type        = string
  description = "Environment name"

  validation {
    condition     = contains(["dev", "staging", "prod"], var.environment)
    error_message = "Environment must be dev, staging, or prod."
  }
}
```

3.3 Using locals for Clean Logic

locals simplify repeated values or derived expressions.

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

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

3.4 Using count and for_each

To dynamically create multiple resources:

With count:

```
resource "aws_subnet" "public" {
  count      = length(var.public_subnet_cidrs)
  cidr_block = var.public_subnet_cidrs[count.index]
  vpc_id     = aws_vpc.main.id
}
```

With for_each:

```
resource "aws_security_group_rule" "ingress" {
  for_each = var.ingress_rules

  type              = "ingress"
  from_port         = each.value.from
  to_port           = each.value.to
  protocol          = "tcp"
  cidr_blocks       = each.value.cidr_blocks
  security_group_id = aws_security_group.main.id
}
```

3.5 Avoiding Hardcoded Values

Always expose values as variables instead of embedding them in resource blocks.

Instead of:

```
cidr_block = "10.0.0.0/16"
```

Use:

```
cidr_block = var.vpc_cidr
```

And define it in variables.tf.

3.6 Managing Defaults with Overridable Values

Define sane defaults in your module to support flexible reuse:

```
variable "instance_type" {
  description = "EC2 instance type"
  type        = string
  default     = "t3.micro"
}
```

Then allow consumers to override only when needed.

4. Versioning and Organizing Modules

4.1 Monorepo vs Polyrepo Approaches

Monorepo: All modules live in one repository.

- **Pros**: Easier cross-module updates, unified versioning, single CI/CD pipeline.
- **Cons**: Larger repo, complex history, permission management might be harder.

Polyrepo: Each module has its own repository.

- **Pros**: Clean isolation, fine-grained version control, easier to open-source/share.
- **Cons**: More complex CI setup, harder to keep shared logic consistent.

Choose based on team size, module count, and deployment frequency.

4.2 Naming Conventions and Directory Organization

Organize modules by domain or service:

```
modules/
├── networking/
├── compute/
├── storage/
├── monitoring/
```

Use consistent naming:

- terraform-<provider>-<service> for registries (e.g., terraform-aws-vpc)
- networking, ec2, rds for internal projects

Document every module with a README.md including:

- Inputs and defaults
- Outputs
- Example usage

4.3 Publishing to Terraform Registry or Git

You can share modules via:

- **Terraform Public Registry**: Add a terraform-<namespace>-<name> pattern and a valid versions.tf.
- **Private GitHub Repo**: Use source = "git::https://github.com/org/module.git?ref=v1.0.0"
- **Private Terraform Registry (TFE/Cloud)**

Ensure you use version tags (v1.0.0, v1.0.1, etc.) for all releases.

4.4 Semantic Versioning and Tagging

Follow **Semantic Versioning**:

- MAJOR: Breaking changes
- MINOR: Backward-compatible new features
- PATCH: Bug fixes only

Tag releases in Git:

```
git tag v1.0.0

git push origin v1.0.0
```

Consumers can pin module versions:

```
module "vpc" {
  source  = "git::https://github.com/org/vpc.git?ref=v1.0.0"
}
```

Avoid using main or latest for production deployments.

4.5 Managing Breaking Changes

- Introduce changes in a new major version.
- Maintain backward compatibility when possible.
- Clearly document deprecations.
- Use default, nullable, or conditional logic to prevent failure in older configs.

Example:

```
variable "enable_dns_hostnames" {
  description = "Enable DNS hostnames in the VPC"
  type        = bool
  default     = false
}
```

5. Using and Managing Remote Modules

5.1 Referencing Remote Modules

Modules can be sourced from:

- **Terraform Registry** (public or private)
- **Git repositories**
- **Local paths**
- **HTTP URLs**
- **S3 buckets** (less common)

Examples:

From the Terraform Registry:

```
module "vpc" {
  source  = "terraform-aws-modules/vpc/aws"
  version = "4.0.0"
  name    = "prod-vpc"
  cidr    = "10.0.0.0/16"
}
```

From Git:

```
module "vpc" {
  source = "git::https://github.com/my-org/terraform-vpc.git?ref=v1.0.2"
}
```

From a local path:

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

5.2 Version Pinning and Upgrades

Always pin specific versions to avoid unintentional updates:

```
version = "4.0.0"
```

To upgrade:

1. Review CHANGELOG for the module.
2. Update the version block.
3. Run terraform init -upgrade to fetch the new version.
4. Validate with terraform plan.

5.3 Overriding Defaults and Providing Custom Configs

Modules expose variables you can override:

```
module "vpc" {
  source  = "terraform-aws-modules/vpc/aws"
  version = "4.0.0"

  name                 = "custom-vpc"
  cidr                 = "10.1.0.0/16"
  enable_dns_hostnames = true

  public_subnets = ["10.1.1.0/24", "10.1.2.0/24"]
}
```

Avoid editing the module’s internal files. If you need to change logic:

- Fork the module and point to your own version
- Or wrap it in your own module and use composition

5.4 Handling Module Outputs

Modules expose outputs that can be referenced like any other Terraform value:

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

You can pass these outputs to other modules or resources:

```
resource "aws_subnet" "subnet" {
  vpc_id     = module.vpc.vpc_id
  cidr_block = "10.1.3.0/24"
}
```

5.5 Keeping Remote Modules Secure and Reliable

- Use **version pinning** to prevent auto-upgrades.
- Prefer **Git tags** over branches for Git-based modules.
- Audit module code before usage, especially from public registries.
- Limit external dependencies by copying or vendoring critical modules if needed.

6. Testing and Validating Modules

6.1 Manual Testing in a Sandbox Environment

Always test modules in isolated environments before using them in production:

- Create a dedicated test project (e.g., test/)
- Use minimal configuration to validate basic functionality
- Run:

```
terraform init
terraform apply
terraform destroy
```

6.2 Validating Syntax and Configuration

Use built-in Terraform commands to ensure correctness:

```
terraform validate    # Check syntax and configuration
terraform fmt -check  # Verify code is formatted
terraform plan        # Preview infrastructure changes
```

Add these checks to your pre-commit hooks or CI pipelines.

6.3 Linting with tflint

tflint helps catch errors and enforce best practices.

Install and run:

```
tflint
```

You can customize rules using .tflint.hcl to match your org's standards.

6.4 Unit Testing with terraform-compliance or OPA

For policy-as-code or unit-like tests, use:

terraform-compliance – behavior-driven compliance:

```
terraform plan -out=tfplan.binary
terraform-compliance -p tfplan.binary -f tests/
```

OPA (Open Policy Agent) – define rules in Rego to ensure compliance.

6.5 Integration Testing with Terratest (Go)

Use Terratest to write automated tests in Go:

Example:

```
func TestVpcModule(t *testing.T) {
  terraformOptions := &terraform.Options{
    TerraformDir: "../examples/vpc-basic",
  }

  defer terraform.Destroy(t, terraformOptions)
  terraform.InitAndApply(t, terraformOptions)

  vpcID := terraform.Output(t, terraformOptions, "vpc_id")
  assert.True(t, strings.HasPrefix(vpcID, "vpc-"))
}
```

Terratest supports parallel test runs, retries, and assertions.

6.6 CI/CD Integration for Terraform Modules

Integrate Terraform workflows with tools like:

- GitHub Actions
- GitLab CI/CD
- CircleCI

Example GitHub Actions job:

```
jobs:
  terraform:
```

```
runs-on: ubuntu-latest

steps:

  - uses: actions/checkout@v3

  - uses: hashicorp/setup-terraform@v3

    with:

      terraform_version: 1.6.2

  - run: terraform init

  - run: terraform validate

  - run: terraform plan
```

7. Managing State and Collaboration

7.1 Understanding Terraform State

Terraform stores infrastructure metadata in a state file (terraform.tfstate):

- Maps real-world resources to config
- Tracks dependencies
- Is critical for Terraform to function correctly

Avoid manual edits and always back up.

7.2 Remote State Storage with Backends

Use backends for state storage and collaboration:

Common backends:

- AWS S3 (with DynamoDB for locking)
- Terraform Cloud/Enterprise
- Azure Blob Storage
- Google Cloud Storage

Example (S3 + DynamoDB):

```
terraform {
  backend "s3" {
    bucket         = "my-terraform-state"
    key            = "networking/vpc.tfstate"
    region         = "us-east-1"
    dynamodb_table = "terraform-locks"
    encrypt        = true
  }
}
```

7.3 Enabling State Locking and Concurrency

Prevent simultaneous changes using state locking:

- **S3 + DynamoDB**: Enables locking

- **Terraform Cloud**: Built-in locking
- **Local backend**: No locking – risky for teams

7.4 Using terraform state for Inspection and Recovery

Useful terraform state commands:

```
terraform state list                    # List tracked resources
terraform state show aws_vpc.main       # View resource details
terraform state rm <address>            # Remove from state
terraform state mv old new              # Rename resources in state
```

7.5 Workspaces for Environment Isolation

Use **workspaces** to manage multiple environments (e.g., dev, staging, prod) with the same codebase.

```
terraform workspace new dev
terraform workspace select dev
terraform apply
```

State files are isolated per workspace.

⚠ Note: Avoid overusing workspaces for unrelated projects.

7.6 Team Collaboration Tips

- Use remote backends
- Enforce Terraform version with .terraform-version or required_version
- Lock modules and provider versions
- Use pull requests and code reviews for all changes
- Store terraform.tfvars or *.auto.tfvars securely (never commit secrets)

8. Real-World Use Cases and Best Practices

8.1 Using Modules for Cross-Team Collaboration

When working in large teams or organizations, modules enable shared resources and configurations. By centralizing infrastructure in reusable modules:

- Teams can standardize infrastructure deployments.
- Avoid duplication of effort and promote code reuse.
- Maintain consistency across environments (e.g., VPC, security groups, IAM roles).

Example:
A centralized **VPC module** can be used across multiple projects and environments.

8.2 Managing Different Cloud Providers with Modules

When working with multiple cloud providers (AWS, Azure, GCP), modules abstract away the provider-specific configurations, allowing a consistent interface for provisioning resources.

For instance, a **networking module** can be made provider-agnostic:

```
# module-networking
variable "cloud_provider" {
  type = string
}

resource "aws_vpc" "main" {
  count = var.cloud_provider == "aws" ? 1 : 0
  cidr_block = var.vpc_cidr
}

resource "azurerm_virtual_network" "main" {
  count = var.cloud_provider == "azure" ? 1 : 0
  address_space = [var.vpc_cidr]
}
```

8.3 Managing State in Multi-Region or Multi-Account Environments

For cross-region or multi-account setups:

- Split state files per region or account.
- Use different backends for each region (e.g., multiple S3 buckets).
- Use IAM roles for secure cross-account Terraform access.

Example:
Deploy a **VPC** in multiple regions with different state files and backend configurations.

```
backend "s3" {
  bucket = "terraform-state-us-west-2"
  key    = "vpc/us-west-2.tfstate"
  region = "us-west-2"
}
```

8.4 Secrets Management

- Avoid hardcoding secrets in Terraform files.
- Use **AWS Secrets Manager**, **Azure Key Vault**, or **HashiCorp Vault** to manage sensitive data.
- Pass secrets via environment variables or external files (terraform.tfvars).

Example of referencing an AWS secret:

```
data "aws_secretsmanager_secret" "db_password" {
  name = "my-database-password"
}

resource "aws_db_instance" "main" {
  password = data.aws_secretsmanager_secret.db_password.secret_string
}
```

8.5 Optimizing for Cost Management

- Use **Terraform Cost Estimation** tools like terraform cost to visualize cost impacts before applying changes.
- Tag resources effectively for cost tracking (e.g., Project, Environment).
- Avoid creating unused resources—use **count** and **for_each** to create resources only when necessary.

8.6 Handling Drift and State Management

Drift happens when resources in the cloud are modified outside of Terraform, causing the state to become inconsistent. Prevent and manage drift by:

- Using **terraform plan** frequently to detect drift.
- Review and fix drift manually or by using terraform refresh.

Example:

```
terraform refresh  # Updates state to reflect the real infrastructure

terraform plan     # Detects changes
```

8.7 Monitoring Infrastructure and Continuous Improvement

- Integrate **monitoring** for your infrastructure (CloudWatch, Azure Monitor, etc.) to track health and performance.
- Use **CI/CD pipelines** to continuously validate and apply Terraform changes.
- Regularly refactor your Terraform code for efficiency, readability, and maintainability.

Conclusion

Building reusable and scalable infrastructure with Terraform is essential for managing cloud resources efficiently and maintaining consistency across environments. By utilizing Terraform modules, version control, state management, and best practices, teams can ensure the long-term sustainability and scalability of their infrastructure.

Key takeaways include:

- **Modular design** helps simplify and standardize infrastructure management.
- **Versioning and organizing** modules ensures smooth upgrades and collaboration.
- **Testing, validation, and CI/CD integration** enable robust and error-free infrastructure deployments.
- **State management** ensures that Terraform can track changes accurately and collaborate effectively in teams.
- **Real-world use cases** and **best practices** emphasize the importance of security, cost optimization, and continuous improvement.

By adhering to these principles, you can create a flexible, maintainable, and secure infrastructure that scales with your organization’s needs. Terraform not only enables consistency across environments but also fosters collaboration and efficiency across development teams.

As you implement these practices in your own workflows, you'll enhance the reliability and agility of your infrastructure management, enabling faster delivery and smoother operational processes.