Infrastructure-as-Code & Terraform

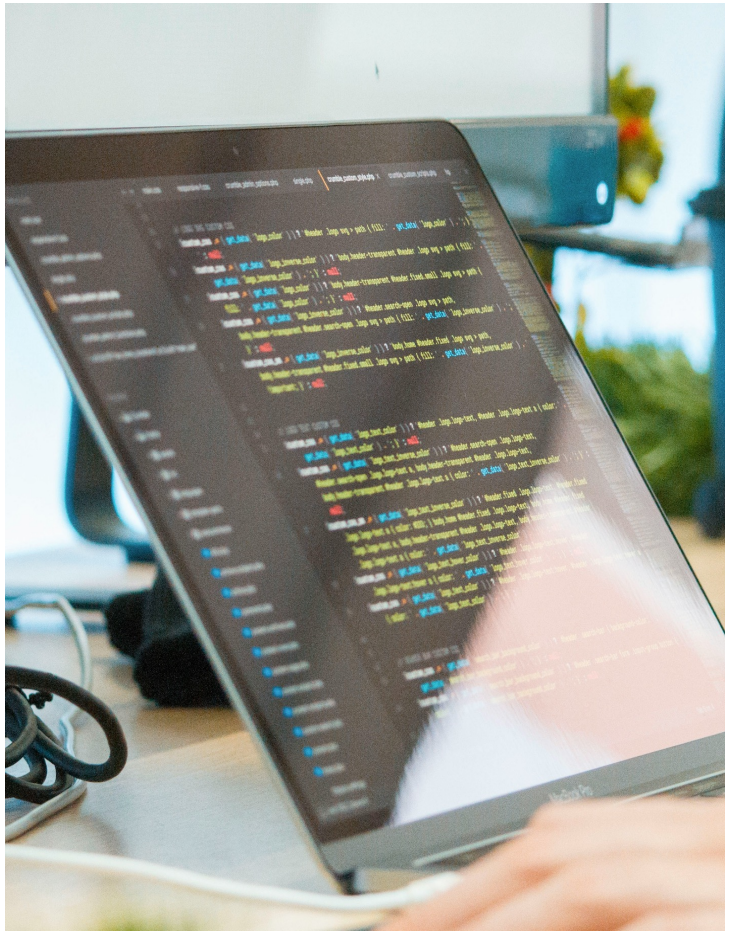
TEKsystems Global Services

WORKFORCE DEVELOPMENT

HCP Backends

Terraform backends decide where your state file is stored. This determines how teams collaborate, how safe your state is, and how Terraform keeps track of real infrastructure.

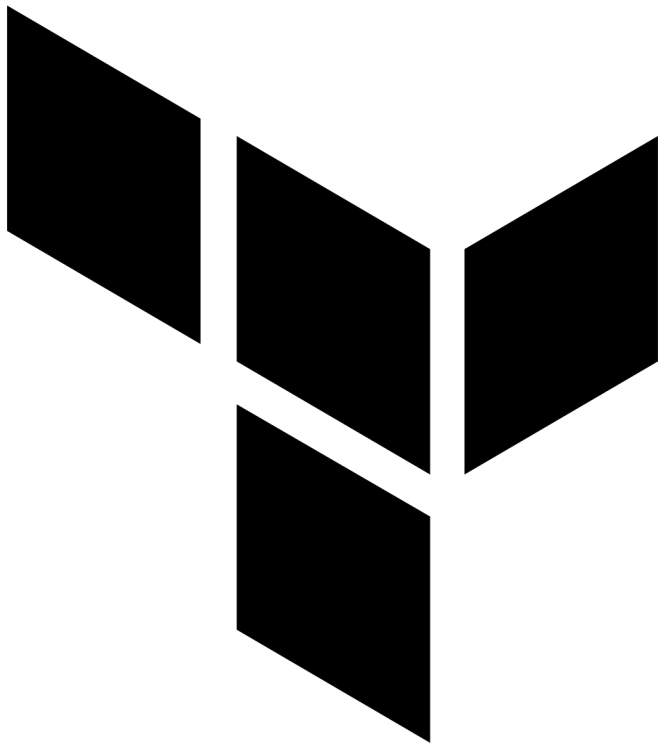
Why Backends Matter?



Terraform uses a state file to understand what exists in your environment. Storing this file in a backend keeps it safe and prevents conflicts when multiple people run Terraform.

- Centralized state for teams
- Better durability and recovery
- Locking to prevent corrupt state

Local vs Remote Backends



Terraform stores state locally by default, but local state is risky for teams. Remote backends move the state into a shared and secure location that everyone can use safely.

- Local is simple but only safe for one person
- Remote enables collaboration
- Remote supports locking, versioning, and encryption

Backend Types

Terraform supports many backend options. Cloud environments often use a remote backend to store the state in a reliable and centralized service.

- S3 backend
- AzureRM backend
- GCS backend
- Consul backend

Terraform Backend

```
terraform {
  backend "s3"{

    bucket    = "bucket-name"
    key       = "prod/terraform.tfstate"
    region    = "us-west-1"
    dynamodb_table  = "var.instance_typ
e"
  }
}
```

The S3 backend uses a bucket for state storage and can use a DynamoDB table for state locking. The bucket and table must already exist before initialization. Locking is optional but strongly recommended for team environments.

DynamoDB creates a lock so only one person runs Terraform at a time

Locking is not required but is a best practice to prevent state corruption

Terraform Backend

```
terraform {
  backend "s3"{

    bucket    = "bucket-name"
    key       = "prod/terraform.tfstate"
    region    = "us-west-1"
    dynamodb_table  = "var.instance_typ
e"
  }
}
```

The backend block defines how Terraform connects to the remote state storage. Each attribute tells Terraform where the state lives and how to protect it.

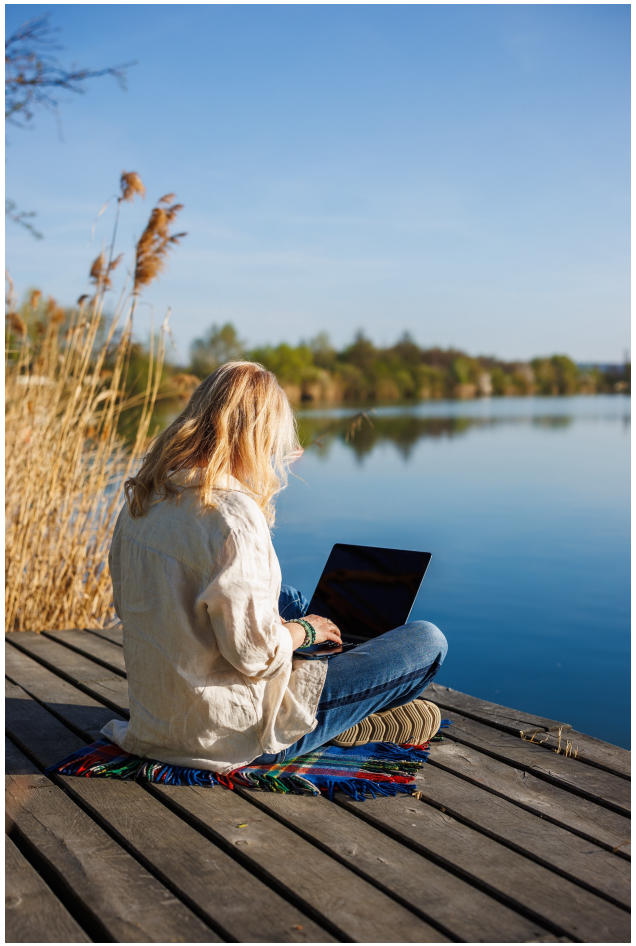
- **bucket** - The S3 bucket that stores the state file
- **key** - The path and file name inside the bucket
- **region** - The AWS region of bucket
- **dynamodb_table** -The table used for optional state locking

State and Infrastructure Drift

Terraform tracks every resource it creates in the state file. The state becomes Terraform’s source of truth for what your cloud environment should look like. When something outside of Terraform changes a resource, the real environment no longer matches the state. This is called drift.

- Drift happens when external changes modify infrastructure
- Terraform detects drift during plan
- Keeping state accurate ensures predictable deployments

What Causes Drift



Drift occurs when real infrastructure changes outside of Terraform's control. These changes move the environment away from the desired state defined in code.

- Manual edits in the cloud console
- Another tool modifying resource
- Autoscaling creating or removing instances
- Failed Terraform runs that partially apply changes
- Missing or deleted state files
- Human error or emergency fixes made outside IaC

Fixing Drift

Terraform can correct drift by comparing real infrastructure with the state file. In many cases Terraform automatically brings the environment back to the desired state, but this is not always what you want. There are cases where you need to adjust or repair state manually.

- Terraform can fix drift by updating resources during apply
- You can untrack a resource with terraform state rm if you want Terraform to ignore it
- You can add an existing resource to state with terraform import to prevent unwanted recreation
- State repairs help avoid surprises during future plans and applies

When to Adjust State

State changes are sometimes needed when production reality moves faster than Terraform. These are common situations where fixing or updating state is appropriate.

•Hot fix changed something in the console

• A resource was created manually and needs to be imported

•Terraform would recreate something you want to keep

• A resource should no longer be managed, so you remove it from state

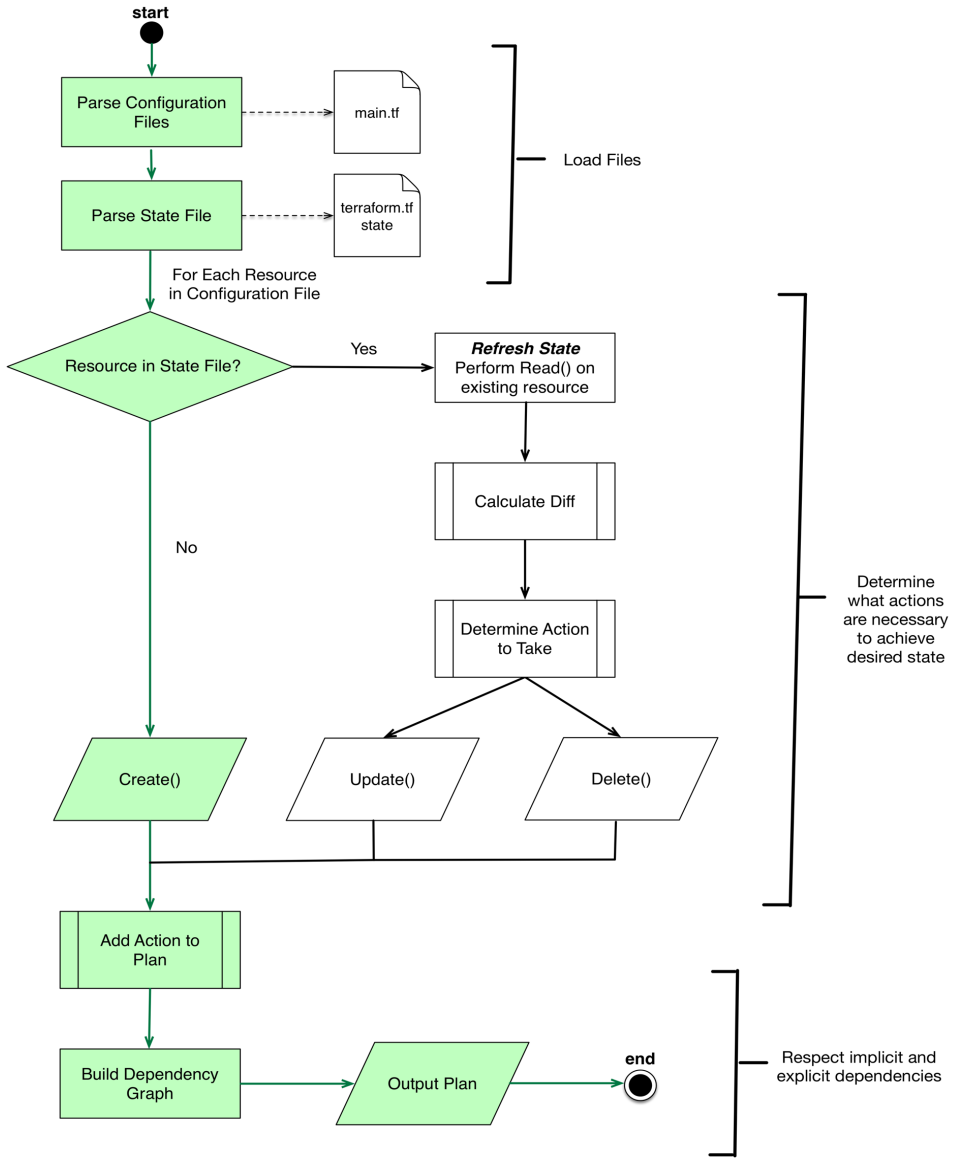
Terraform State Management

When Terraform creates a new infrastructure object represented by a resource block, the identifier for that real object is saved in Terraform's state, allowing it to be updated and destroyed in response to future changes.

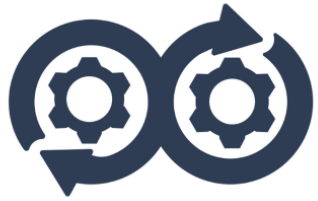
Terraform State Management

Resource blocks that already have an associated infrastructure object in the state, Terraform compares the actual configuration of the object with the arguments given in the configuration and, if necessary, updates the object to match the configuration.

Terraform State Management



Lab: Terraform Backends



Terraform Data Sources

Data sources let Terraform read information from the cloud without creating anything. They allow your configuration to reference existing values so you can build resources dynamically and avoid hard-coding.

- Used to look up existing VPCs, AMIs, subnets, IAM roles
- Helpful when mixing IaC with existing infrastructure
- Returned values can be used anywhere in your configuration

Use Cases for Data Sources

Data sources let Terraform pull in information that already exists in the cloud so you can reference it safely without recreating anything. This is useful when your code needs to adapt to the real environment.

- Use the caller identity or role name of the person running Terraform for tagging
- Look up the ID of an existing Lambda function
- Read an existing IAM role so you can update its trust policy
- Retrieve VPCs, subnets, or security groups created outside Terraform

AWS Data Source Example

```
Data "aws_caller_identity" "current" {}

Resource "aws_s3_bucket" "logs" {

  bucket   = "bucket-name"
  tags = {
    bucket = data.aws_caller_identity.current
.arn
  }

}
```

This example uses a data source to read information about the current AWS identity. Terraform does not create anything from the data block. It only reads the account details so we can use them in resource names and tags.

The data source reads the current AWS account and user or role. The S3 bucket then uses that data to build a unique name and helpful tags.

AWS Data Source Example

```
data "aws_default_vpc" "default" {}

data "aws_default_subnet" "default_az1" {
  availability_zone = "us-east-1a"
}

resource "aws_instance" "web"
{
  subnet_id = aws_default_subnet.default_az1
}
```

Data sources let Terraform detect whether a default VPC exists in the account. This is useful when your configuration depends on network resources that may or may not be present.

- You can verify a default VPC exists before deploying

- You can look up default subnets to attach resources

AWS Data Source Example

```
data "<TYPE>" "<LABEL>" {
   <PROVIDER-SPECIFIC ARGUMENTS>
   count = <NUMBER>      # `count` and `for_each` are mutually exclusive
   depends_on = [ <RESOURCE.ADDRESS.EXPRESSION> ]
   for_each = {          # `for_each` and `count` are mutually exclusive
      <KEY> = <VALUE>
   }
   for_each = [       # `for_each` accepts a map or a set of strings
    "<VALUE>",
    "<VALUE>"
   ]
   provider = <REFERENCE.TO.ALIAS>
   lifecycle {
      precondition {
         condition = <EXPRESSION>
         error_message = "<STRING>"
    }
    postcondition {
      condition = <EXPRESSION>
      error_message = "<STRING>"
    }
  }
}
```

The code block shows every argument that can be used with the data block. Note that availability zone and other arguments are provider specific.

- Many terraform projects have multiple providers. Use the provider configuration to specify a provider

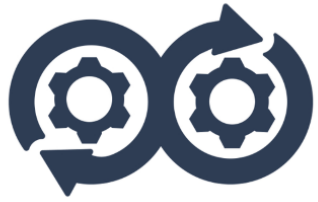
https://developer.hashicorp.com/terraform/language/block/data#complete-configuration

Data Sources Evaluate at Plan Time

The values returned by a data source are pulled during the plan phase.

- If something changes in AWS between plan and apply, Terraform may complain
- It helps explain why some edge cases happen
- Reinforces why drift can matter

Lab: Terraform Data



Don't Repeat Yourself (DRY)

DRY is a principle that discourages repetition, and encourages modularization, abstraction, and code reuse. Applying it to Terraform, using modules is a big step in the right direction.

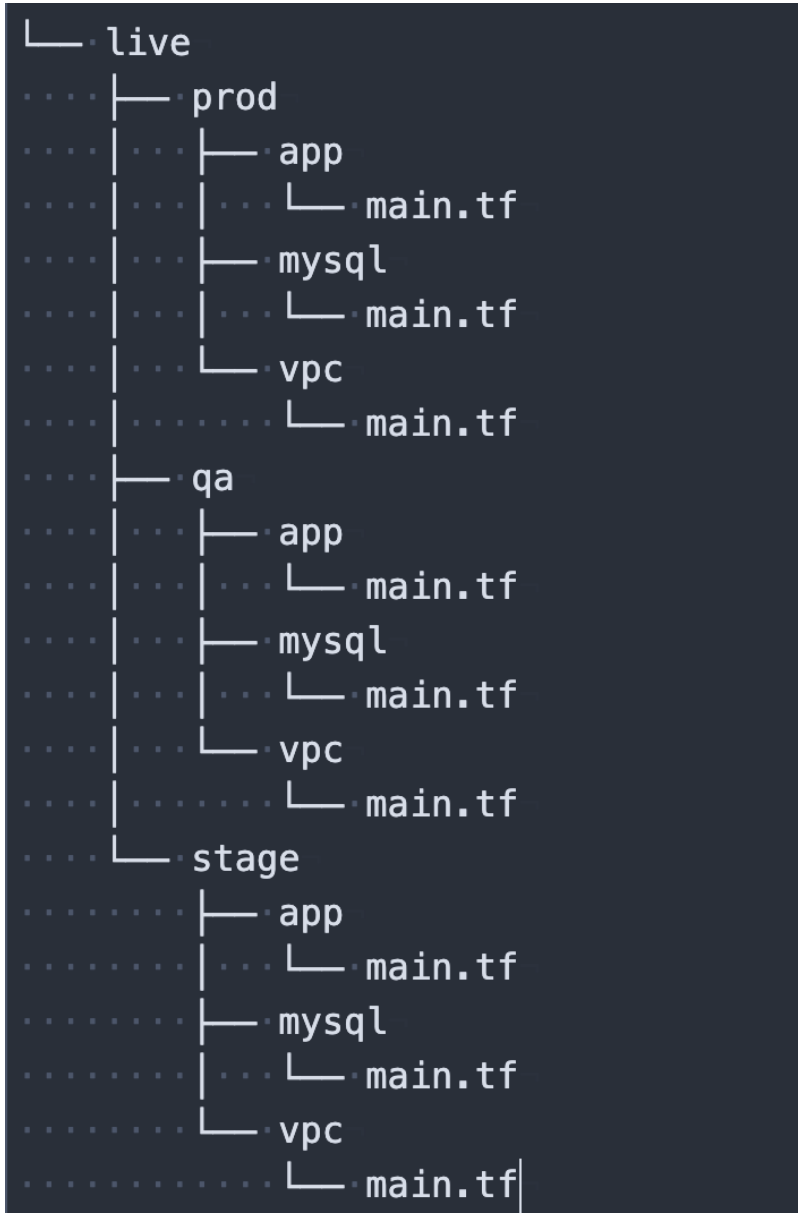
However, repetitions still happen. You may end up having virtually the same code in different environments, and when you need to make one change, you have to make that change many times.

Don't Repeat Yourself (DRY)

This problem can be addressed a few ways. One approach is to create a folder for shared or common files, and then create symlinks to these files from each environment. This way, you can make a change to the common file(s) once and it is applied in all the environments.

Don't Repeat Yourself (DRY)

Common directory structure for managing three environments (prod, qa, stage) with the same infrastructure (an app, a MySQL database and a VPC)



DRY Principle with Pipeline-Driven Environments

```
terraform/
├── main.tf           # Main configuration
├── variables.tf      # Variable definitions
├── outputs.tf        # Output definitions
├── terraform.tfvars  # Default values
└── environments/     # Environment-specific values
    ├── dev.tfvars
    ├── staging.tfvars
    └── prod.tfvars
```

While environment-specific directories are common, pipeline-driven environments with a single configuration provide better state management and control.

- Single configuration path for all environments
- Pipeline controls environment selection
- Automated state file management
- Reduced configuration duplication

Pipeline-Driven Environment Management

```
variables:
  TF_STATE_PREFIX:
${CI_PROJECT_NAME}/${CI_ENVIRONMENT_NAME}


init:
  script:
    - |
    # Configure backend for this environment
    cat > backend.hcl <<EOF
    bucket = "$TF_STATE_BUCKET"
    prefix = "$TF_STATE_PREFIX"
    EOF
    # Initialize with environment-specific
backend
    terraform init -backend-
config=backend.hcl
```

Using GitLab CI/CD pipelines to manage environments provides better control and state isolation.

- Pipeline selects environment configuration
- Automatic backend state path generation
- Environment-specific service accounts
- Controlled access through pipeline

Don't Repeat Yourself (DRY)

There is also an open source tool, Terragrunt which solves the same problem in a different way. It is a wrapper around the Terraform CLI commands, which allows you to write your Terraform once, and then in a separate repository define only input variables for each environment - no need to repeat Terraform code for each environment. Terragrunt is also handy for orchestrating Terrfaorm in CICD pipelines for multiple separate projects.

Standard Terraform Module Structure

Terraform modules are reusable collections of configuration files that follow a recommended structure.

The only required element is the root module, which consists of Terraform files in the root directory. Adhering to the standard structure is not mandatory, but it greatly improves documentation, usability, and compatibility with Terraform tooling. This structure also helps with automatic documentation generation and module registry indexing.

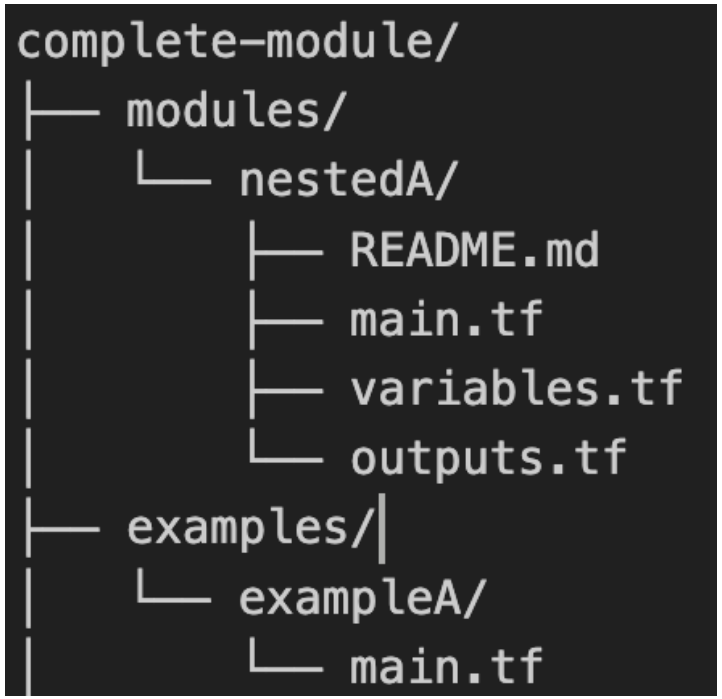
Minimal Recommended Module Layout

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

A minimal Terraform module should include the following files to ensure clarity, reusability, and ease of use:

- README.md — Provides a description of the module, its purpose, and basic usage instructions.
- main.tf — The primary entrypoint for resource creation. This is where you define the main resources that the module manages.
- variables.tf — Declares all input variables for the module. Each variable should have a clear description, making it easier for users to know what values they need to provide.
- outputs.tf— Defines the outputs that the module will return. Outputs allow users to access information about resources created by the module, such as IDs or IP addresses.

Advanced Structure: Nested Modules



For more complex modules, you can organize your code with nested modules and usage examples.

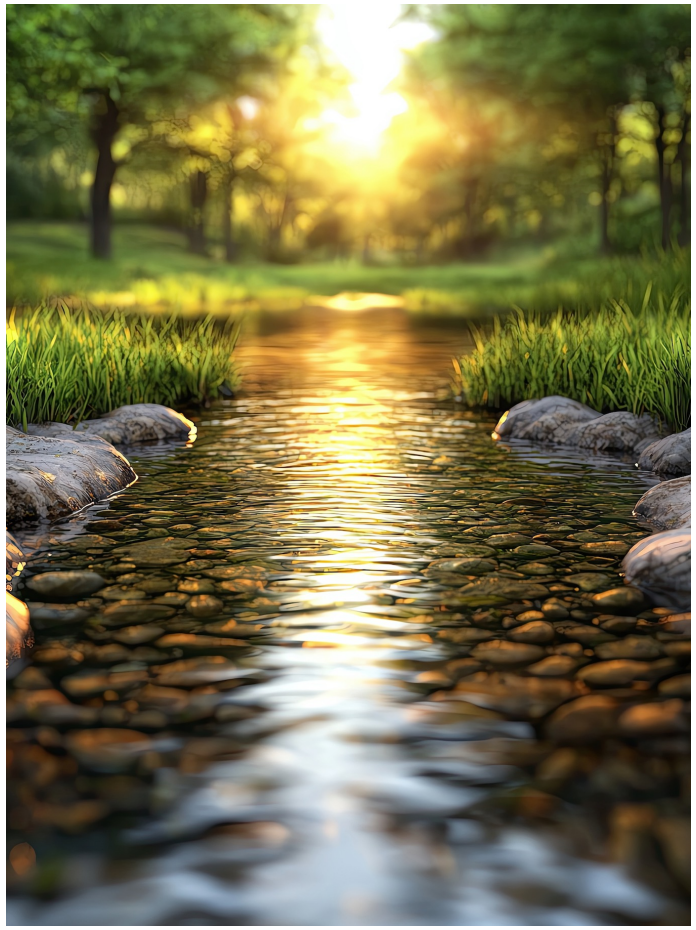
Nested modules are placed in a `modules/` subdirectory, each with its own configuration files and README. Usage examples should be placed in an `examples/` directory, with each example in its own folder. Including a LICENSE file is also recommended for public modules.

Terraform Module Design Overview

Terraform modules are self-contained, reusable pieces of infrastructure-as-code that help abstract complexity and promote best practices like DRY (Don't Repeat Yourself).

Good module design starts with scoping requirements into focused, opinionated modules that do one thing well. When designing a module, consider encapsulation (group resources that are always deployed together), privilege boundaries (keep resources with different access needs separate), and volatility (separate long-lived from short-lived infrastructure). Aim for a minimum viable product that covers most use cases, exposes useful outputs, and is well-documented. This approach makes modules easier to use, share, and maintain across teams and projects.

Use Source Control to Track Modules



A Terraform module should follow all good code practices, and source control is essential for this. Place each module in its own repository to manage release versions, enable collaboration, and maintain an audit trail of changes. Tag and document all releases to the main branch, using a CHANGELOG and README at minimum.

Code review all changes before merging to main, and encourage users to reference modules by tag for stability. Assign an owner to each module, and ensure only one module exists per repository—this supports idempotency, library-like usage, and is required for private registry compatibility.

Develop a Module Consumption Workflow

Define and publicize a repeatable workflow for teams consuming your modules. This workflow should be shaped by user requirements and make it easy for teams to adopt modules consistently. HCP Terraform offers tools like the private Terraform registry and configuration designer, which provide structure for module collaboration and consumption.

These tools simplify module discovery, usage, and integration, making modules more accessible and reducing onboarding friction for new users.

Make Modules Easy and Secure to Use

- Private Terraform registry: Offers a searchable, filterable way to manage and browse modules.
- UI: The Terraform Enterprise UI lowers the barrier for new users.
- Configuration designer: Provides interactive documentation and advanced autocompletion, helping users discover variables and outputs quickly.
- Devolved security: Repository RBAC allows teams to manage their own modules securely.
- Policy enforcement: Use Sentinel to enforce that only approved modules from the private registry are used, supporting compliance and governance.

Using a Module

```
module "ec2_server"{
  source = "./modules/ec2"

  instance_type= "bucket-name"
  enable_backup = "bucket-name"
}
```

Modules let you group Terraform code into reusable units. They make your configurations cleaner and allow you to share common patterns across environments.

The module block calls a folder of Terraform code and passes in the values it needs. This keeps your main configuration small and organized.

How Modules Receive Input

Terraform/main.tf

```
module "ec2_server"{
  source = "./modules/ec2"

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

Terraform/modules/ec2/variables.tf

```
Variable "instance_type"{
  type        = string
  description = "Instance type for modules ec2"
  default     = "t2.micro"
}

Variable "enable_backup"{
  type        = bool
  description = "set to true for backups"
  default     = false
}
...
```

The source argument tells Terraform where the module code is located. The attributes inside the module block are inputs that match the variables defined in the module's variables.tf file.

- **source** points to a folder, Git repo, registry module, or local path
- The other arguments must match the module's variable names
- These values flow into the module just like passing arguments into a function

Using Module Outputs

Terraform/main.tf

```
module "ec2_server"{
  source = "./modules/ec2"
}

resource "aws_eip" "server_ip" {
  instance = moduel.ec2_sercer.instance_id
}
...
```

Terraform/modules/ec2/outputs.tf

```
Output "instance_id"{
  value         = saws_isntance.server.id
  description = "ID for EC2 instance"
}
...
```

A module's internal resources cannot be accessed directly. Terraform only allows the calling configuration to read values that the module explicitly outputs. This keeps modules clean and prevents hidden dependencies.

- You can reference a module value only if it is defined in the module's outputs.tf
- Outputs act like return values from a function

Example:
`module.ec2_server.instance_id` works because the module exposes instance_id in outputs.tf.

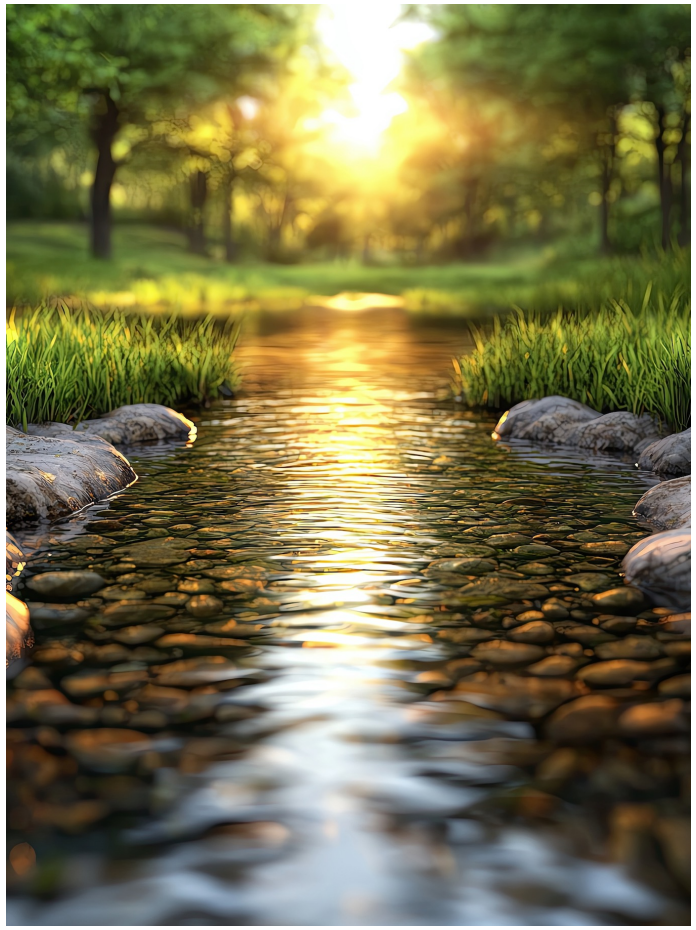
Why Module Outputs Matter

Modules hide their internal resources. If you want a parent configuration to use values from inside a module, those values must be exposed through outputs. Outputs act like return values.

- Use outputs so parent modules can access IDs, ARNs, names, and other values
- Parent modules can only read what the child module explicitly outputs
- If you want to view a module value after apply, the parent must output it as well
- This keeps modules clean, predictable, and reusable

Outputs = the bridge between module internals and the parent configuration.

Variables and Modules



Command-line variables apply only to the parent module. Terraform does not send -var values directly into child modules. You must pass values into modules manually through the module block.

- `terraform apply -var="env=prod"` only sets variables in the root (parent) module
- Child modules do not receive those values automatically
- You must pass them explicitly inside the module block
- Parent variables can be forwarded to child modules by referencing them

Lab: Terraform modules