
PLURALSIGHT

Use Case

The development team needs multiple identical environments (e.g., for Dev, QA and Production). How would you ensure each environment is consistent using Terraform?

Introduction

To ensure that multiple identical environments (such as Dev, QA, and Production) are consistent using Terraform, you can combine **Terraform Cloud** and **Terraform Workspaces** by following best practices for environment isolation, state management, and collaboration.

1. Create Workspaces

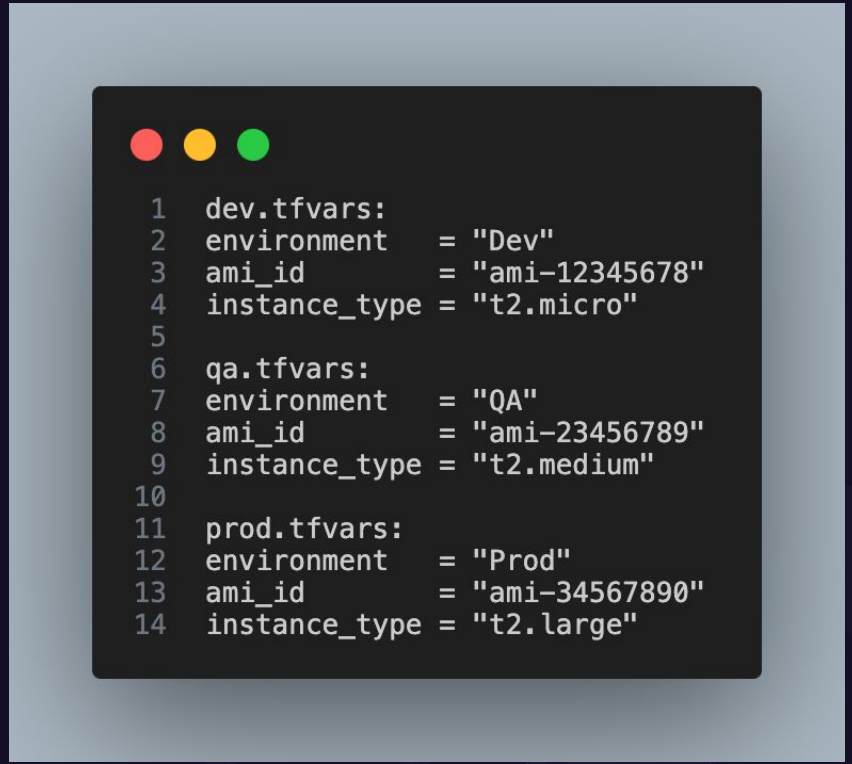
- Workspaces are simply referring to environments.
- Terraform workspaces allow you to use a single configuration with different states for multiple environments (e.g., Dev, QA, Prod).
- Each workspace manages its own state file, ensuring that environments are isolated from each other and infrastructure changes are kept separate while using the same Terraform configurations.
- This approach helps manage state files independently for each environment.

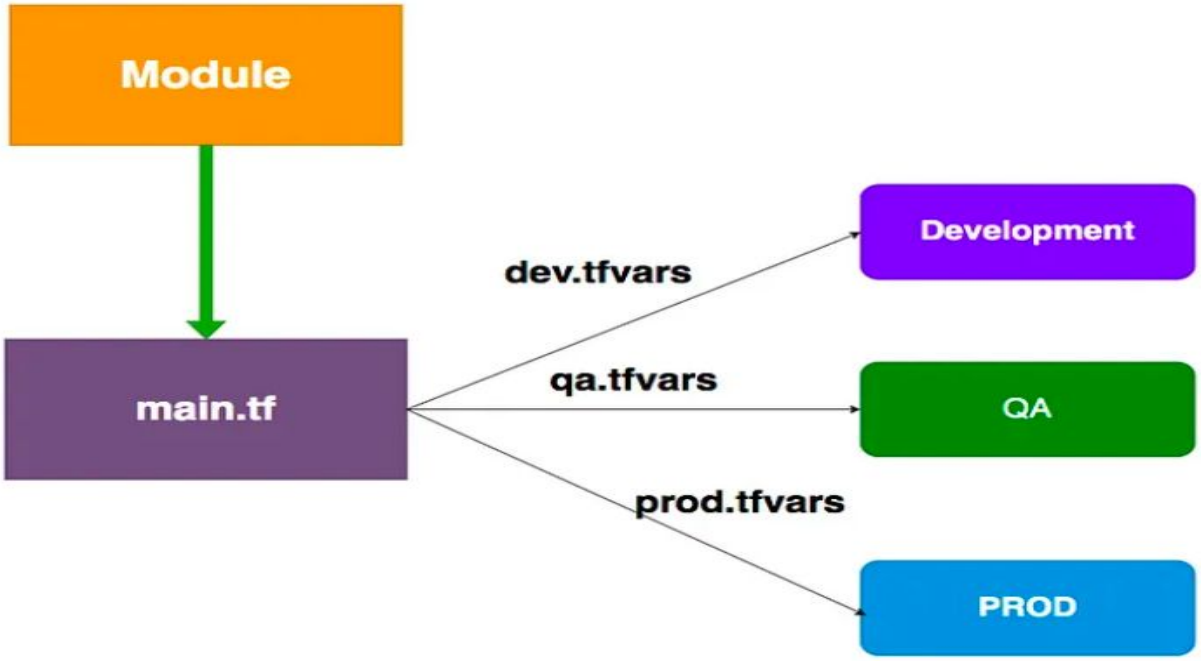
2. Define Infrastructure in Modules

- **Modules** in Terraform allow you to define reusable code that can be shared across environments.
- **Create reusable modules** for common resources like VPC, EC2 instances, RDS, etc.
- Each environment can then instantiate these modules with environment-specific configurations.

3. Use Environment-Specific Variable files

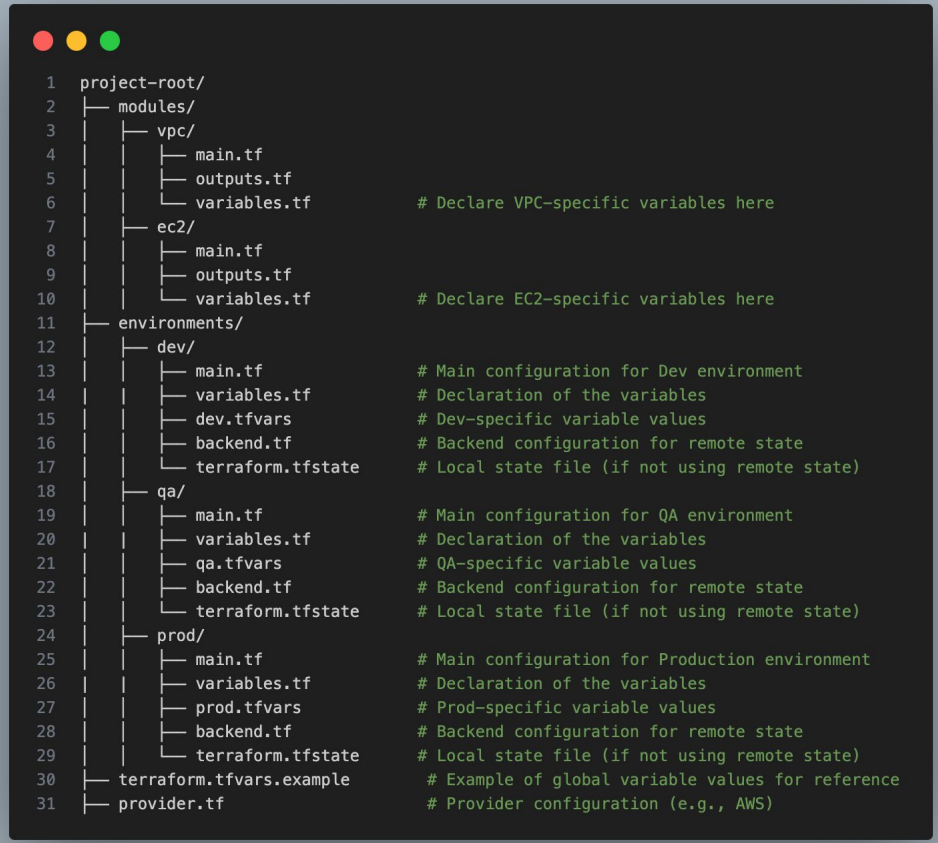
- By using **variables** in Terraform, you can parameterize environment-specific details such as the region, instance types, or scaling parameters while reusing the same base infrastructure configuration.





Multiple Environments

Sample Project Structure



```
project-root/
├── modules/
│   ├── vpc/
│   │   ├── main.tf
│   │   ├── outputs.tf
│   │   └── variables.tf          # Declare VPC-specific variables here
│   ├── ec2/
│   │   ├── main.tf
│   │   ├── outputs.tf
│   │   └── variables.tf          # Declare EC2-specific variables here
├── environments/
│   ├── dev/
│   │   ├── main.tf               # Main configuration for Dev environment
│   │   ├── variables.tf          # Declaration of the variables
│   │   ├── dev.tfvars            # Dev-specific variable values
│   │   ├── backend.tf            # Backend configuration for remote state
│   │   └── terraform.tfstate     # Local state file (if not using remote state)
│   ├── qa/
│   │   ├── main.tf               # Main configuration for QA environment
│   │   ├── variables.tf          # Declaration of the variables
│   │   ├── qa.tfvars             # QA-specific variable values
│   │   ├── backend.tf            # Backend configuration for remote state
│   │   └── terraform.tfstate     # Local state file (if not using remote state)
│   ├── prod/
│   │   ├── main.tf               # Main configuration for Production environment
│   │   ├── variables.tf          # Declaration of the variables
│   │   ├── prod.tfvars           # Prod-specific variable values
│   │   ├── backend.tf            # Backend configuration for remote state
│   │   └── terraform.tfstate     # Local state file (if not using remote state)
├── terraform.tfvars.example       # Example of global variable values for reference
├── provider.tf                    # Provider configuration (e.g., AWS)
```

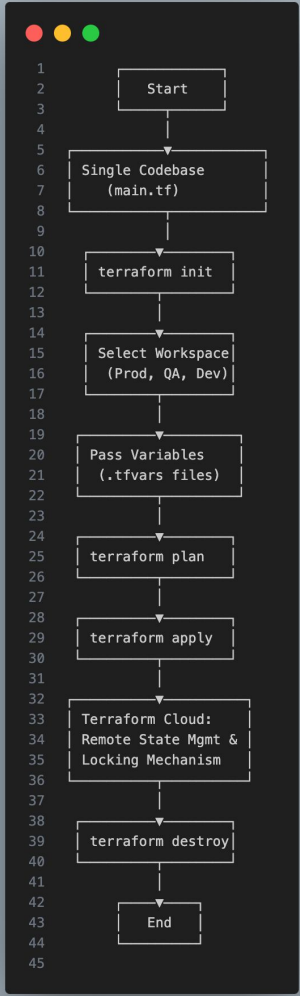
Usage Example

```
- For local development with the *Dev* environment:
cd environments/dev                    # Navigate to the environment
terraform init                         # Initialize the environment
terraform workspace new dev            # Create workspace (run only once) or
terraform workspace select dev         # Select existing workspace

terraform plan -var-file=dev.tfvars    # Plan changes
terraform apply -var-file=dev.tfvars   # Apply changes
terraform destroy -var-file=dev.tfvars # Destroy resources if needed


Similarly, for QA or Prod, you can switch to the respective directories

This sequence ensures that you have a clear flow for initializing, planning,
applying, and destroying Terraform resources for different environments and
workspaces.
```



Sample code for Reusable Modules

```
1. Reusable Modules
a. VPC Module (modules/vpc/main.tf)
variable "cidr_block" {
  type    = string
  default = "10.0.0.0/16"
}
variable "environment" {
  type = string
}
resource "aws_vpc" "main" {
  cidr_block = var.cidr_block
  tags = {
    Name = "${var.environment}-vpc"
  }
}
b. VPC Outputs (modules/vpc/outputs.tf)

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

```
c. EC2 Module (modules/ec2/main.tf)
variable "instance_type" {
  type    = string
  default = "t2.micro"
}
variable "environment" {
  type = string
}
resource "aws_instance" "app" {
  ami           = "ami-0c55b159cbfafe1f0"
  instance_type = var.instance_type
  tags = {
    Name = "${var.environment}-app-instance"
  }
}
d. EC2 Outputs (modules/ec2/outputs.tf)
output "instance_id" {
  value = aws_instance.app.id
}
```

4. Use Terraform Cloud

Leverage Terraform Cloud's Remote State Management and Role Based Access and Policy Enforcement.

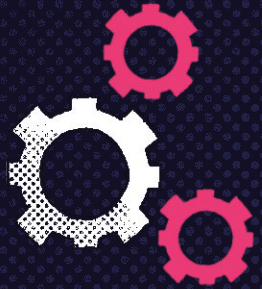
- Terraform Cloud will store the state files for each workspace (Dev, QA, Prod) securely and with encryption at rest.
- It provides a **locking mechanism** to ensure only one process can modify the state at a time, preventing corruption.
- **Role-based Access Control (RBAC)**: Control access to workspaces (Dev, QA, Prod) so that only authorized personnel can make changes.
- **Policy Enforcement**: Use **Sentinel** in Terraform Cloud to enforce policies like requiring approvals before changes are made in production.

Benefits of Combining Terraform Workspaces and Cloud

- **Consistency**: Using a single codebase for all environments ensures that infrastructure changes are consistent across Dev, QA, and Production.
- **Isolation**: Workspaces isolate each environment's state file, so changes in one environment do not impact others.
- **Collaboration**: Terraform Cloud's remote state management and locking mechanism enable teams to collaborate effectively without overwriting each other's changes.
- **Security**: Terraform Cloud provides secure storage with encryption at rest for state files, which may contain sensitive information like credentials and resource identifiers.
- **Scalability**: This approach scales easily, allowing you to add new environments or modify existing ones by reusing the same code and adjusting variable files.

Summary

- **Use Workspaces to isolate environments (Dev, QA, Prod) but maintain a single codebase**
- **Reusable modules (VPC, EC2)** allow code reuse.
- **Environment-specific configurations** allow you to define settings like subnets, instance sizes, etc., for each environment.
- **Leverage Terraform Cloud for secure remote state management with locking and Role-based access control(RBAC) to prevent conflicts and unauthorized access.**
- **Enforce policies using Terraform Cloud's built-in Sentinel policies to ensure only approved changes are applied in production environments.**



THANK YOU