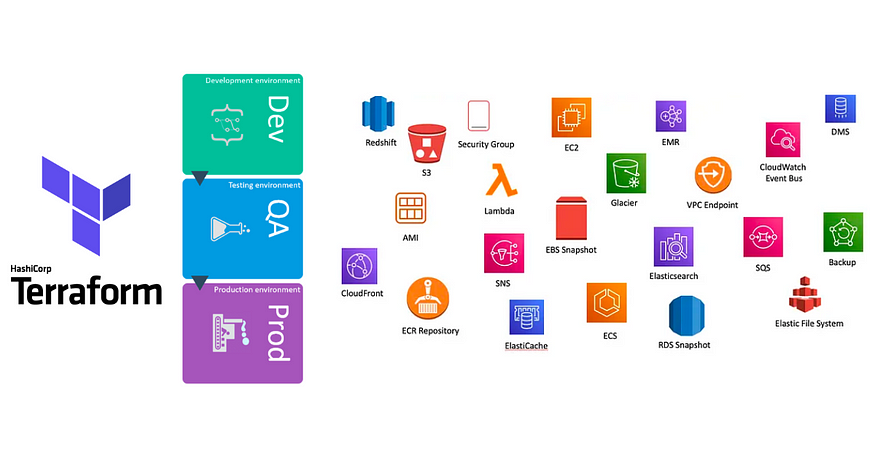
**Terraform manage multiple environments**

How to manage TF multiple environments in your projects

When using Terraform to manage infrastructure, it is common to have multiple environments, such as development, staging, and production. Each environment may have its own set of resources and configurations. To keep the infrastructure code organized and maintainable, it is helpful to use modules.

Modules in Terraform allow you to encapsulate and reuse configurations. By creating modules for different components of your infrastructure, you can easily manage and maintain them across multiple environments.

The solution of using environment directory names with modules involves creating a directory for each environment and organizing the infrastructure code within those directories. For example, you might have directories named “dev”, “staging”, and “prod” for your development, staging, and production environments respectively.



**Table of Contents**

· [What is Terraform?](https://medium.com/@b0ld8/terraform-manage-multiple-environments-63939f41c454#26b7)  
· [Infrastructure for Multiple Environments](https://medium.com/@b0ld8/terraform-manage-multiple-environments-63939f41c454#f409)  
∘ [Requirements](https://medium.com/@b0ld8/terraform-manage-multiple-environments-63939f41c454#43e7)  
· [SOLUTIONS](https://medium.com/@b0ld8/terraform-manage-multiple-environments-63939f41c454#e652)  
·· [1. Terraform workspaces:](https://medium.com/@b0ld8/terraform-manage-multiple-environments-63939f41c454#a428)  
·· [2. Environment-specific directories:](https://medium.com/@b0ld8/terraform-manage-multiple-environments-63939f41c454#538e)  
·· [3. Third-party tools:](https://medium.com/@b0ld8/terraform-manage-multiple-environments-63939f41c454#eb69)  
∘ [Scenario Approaches:](https://medium.com/@b0ld8/terraform-manage-multiple-environments-63939f41c454#b22d)  
∘ [1. Terraform Workspaces:](https://medium.com/@b0ld8/terraform-manage-multiple-environments-63939f41c454#5a6c)  
∘ [2. Environment-specific directories:](https://medium.com/@b0ld8/terraform-manage-multiple-environments-63939f41c454#3606)  
· [PRACTICAL EXAMPLE](https://medium.com/@b0ld8/terraform-manage-multiple-environments-63939f41c454#8c37)  
·· [1. Terraform workspaces:](https://medium.com/@b0ld8/terraform-manage-multiple-environments-63939f41c454#712c)  
·· [2. Environment-specific directories:](https://medium.com/@b0ld8/terraform-manage-multiple-environments-63939f41c454#582a)  
· [Conclusion](https://medium.com/@b0ld8/terraform-manage-multiple-environments-63939f41c454#1237)  
· [References](https://medium.com/@b0ld8/terraform-manage-multiple-environments-63939f41c454#024e)  
· [Source code](https://medium.com/@b0ld8/terraform-manage-multiple-environments-63939f41c454#1392)  
· [About B8](https://medium.com/@b0ld8/terraform-manage-multiple-environments-63939f41c454#0797)

Managing infrastructure across multiple environments can be a daunting task, especially when changes need to be made consistently and reliably. This is where Terraform, an open-source infrastructure as code tool, comes to the rescue. In this blog post, we will explore the power of Terraform and how it simplifies managing multiple environments for your infrastructure.

**What is Terraform?**

Terraform is an open-source Infrastructure as Code (IaC) tool developed by HashiCorp. It allows users to define and manage infrastructure resources in a declarative way, using a simple and easy-to-understand language called HashiCorp Configuration Language (HCL).

With Terraform, you can describe the desired state of your infrastructure, including servers, networks, storage, databases, and more, in a configuration file. This file specifies the resources you want to create, their relationships, and any additional configurations or dependencies.

Once you have defined your infrastructure’s configuration, you can use Terraform to orchestrate the provisioning and management of those resources across various cloud platforms like AWS, Azure, Google Cloud, or on-premises data centers.

**Infrastructure for Multiple Environments**

Terraform’s purpose is to provide one workflow to provision any infrastructure. In this section, we’ll show you our recommended practices for organizing Terraform usage across a large organization. This is the set of practices that we call “collaborative infrastructure as code.”

**Requirements**

Managing multiple environments with Terraform can be a complex task, but it offers significant benefits in terms of infrastructure consistency and reproducibility. However, there are challenges that arise when working with multiple environments, such as development, staging, and production.

Imagine you have to manage infrastructure using Terraform that utilizes:

1. The need for separate configurations for each environment.
2. Coordinating dependencies between environments, such as staging relying on resources provisioned in development, can be cumbersome without proper tooling or practices.
3. Maintaining separate state files for each environment is crucial to ensure proper isolation and management, preventing conflicts and resource issues.
4. Updating infrastructure code across multiple environments requires careful planning, validation, and a clear process to minimize errors and disruptions during promotion.

In a multi-environment setup, state files play a vital role in maintaining consistency, facilitating collaboration, and managing dependencies. By following best practices for managing state files, such as using remote storage and isolating environments, you can enhance the reliability and scalability of your infrastructure provisioning process. Remember, treating state files as a critical component of your infrastructure-as-code workflow is essential for smooth and efficient management of multiple environments with Terraform.

**SOLUTIONS**

When working with multiple environments in Terraform, there are several solutions to consider:

* Leverage Terraform workspaces, which allow you to create separate environments within a single Terraform configuration
* Use environment-specific directories, where you maintain separate directories for each environment and its respective Terraform configurations.
* An external tool can be utilized to handle environment-specific variables or configurations, enhancing flexibility and reducing the need for duplicating code.

**1. Terraform workspaces:**

Terraform CLI workspaces are associated with a specific working directory and isolate multiple state files in the same working directory, letting you manage multiple groups of resources with a single configuration. The Terraform CLI does not require you to create CLI workspaces. Refer to Workspaces in the Terraform Language documentation for more details.

Terraform workspaces allow you to manage multiple instances of your infrastructure within a single Terraform configuration. Each workspace is an independent environment where you can maintain separate sets of resources.

/  
├── provider.tf  
├── variables.tf  
├── main.tf  
├── ec2.tf  
└── vpc.tf

When you start using Terraform, a default workspace called “default” is automatically created. This is where your initial infrastructure is defined unless you explicitly create and switch to a different workspace.

Let’s take a look at the options available to us in the help:

terraform workspace --help  
Usage: terraform [global options] workspace

new, list, show, select and delete Terraform workspaces.Subcommands:  
 delete Delete a workspace  
 list List Workspaces  
 new Create a new workspace  
 select Select a workspace  
 show Show the name of the current workspace

Workspaces should have unique names within a Terraform configuration. When you switch workspaces, Terraform automatically updates the state file to reflect the resources and configurations specific to the selected workspace.

**Workspace interpolation:** In Terraform, workspace interpolation allows you to dynamically reference the current workspace or workspace-specific information within your configuration files. It provides a way to create conditional logic or dynamically generate values based on the workspace being used.

Syntax: terraform.workspace   
Examples:   
- Naming:  
 resource "aws\_instance" "example" {  
 instance\_type = "t2.micro"  
 tags = {  
 Name = "example-${terraform.workspace}"  
 }  
 }  
- Condition:   
 variable "region" {  
 type = string  
 default = terraform.workspace == "production" ? "us-west-2" : "us-east-1"  
 }  
- Variables retrieve:  
 variable "my\_var" {  
 type = map(string)  
 default = {  
 dev = "this is dev"  
 staging = "this is staging"  
 prod = "this is production"  
 }  
 }  
 Name = var.my\_var[terraform.workspace]

As you see, managing workspace-specific variables across multiple environments can become complex and error-prone. It requires careful coordination and documentation to ensure that the correct variables are used for each workspace.

**Separate Terraform Configurations:**

Managing multiple projects or repositories with separate configurations and different requirements, such as the need for a load balancer in the production environment but not in the development environment, can be challenging with Terraform workspaces alone. While workspaces are designed to manage variations within a single configuration, they may not be sufficient for managing completely separate configurations.

Pros

* Allows you to manage multiple environments (e.g., dev, staging, production) using a single Terraform configuration.
* Provides a simple and built-in way to switch between workspaces, making it easier to manage state files and resources.
* Allows you to isolate resources and variables specific to each environment, reducing the chance of misconfiguration.

Cons

* Workspaces can add complexity, especially when managing multiple environments with different configurations.
* Workspaces do not provide complete isolation, as they share the same backend state file. This can lead to potential issues if not managed properly.
* Limited support for complex scenarios, such as managing multiple providers or sharing resources across workspaces.

**2. Environment-specific directories:**

When using Terraform to manage infrastructure, it is common to have multiple environments, such as development, staging, and production. Each environment may have its own set of resources and configurations. To keep the infrastructure code organized and maintainable, it is helpful to use modules.

Modules in Terraform allow you to encapsulate and reuse configurations. By creating modules for different components of your infrastructure, you can easily manage and maintain them across multiple environments.

The solution of using environment directory names with modules involves creating a directory for each environment and organizing the infrastructure code within those directories. For example, you might have directories named “dev”, “staging”, and “prod” for your development, staging, and production environments respectively.

/  
├── environments/  
│ ├── staging/  
│ │ ├── provider.tf  
│ │ ├── variables.tf  
│ │ └── main.tf  
│ └── prod/  
│ ├── providers.tf  
│ ├── variables.tf  
│ / └── app.tf  
└── modules/  
 ├── ec2/  
 ├── vpc/  
 │ ├── main.tf  
 │ └── variables.tf  
 └── application

**Terraform modules**:

Terraform modules are reusable units of infrastructure code that encapsulate resources and configurations. They promote reusability by allowing you to define infrastructure components once and reuse them across different projects or environments. Modules simplify infrastructure management by encapsulating the configuration details of a specific component and providing a higher level of abstraction. They enable collaboration by allowing teams to share and reuse modules, promoting standardization and scalability. With modules, you can organize and maintain infrastructure code in a modular and scalable manner, making it easier to adopt infrastructure-as-code practices and achieve infrastructure automation.

Example: In staging environment, we need set up a VPC (Virtual Private Cloud):

### environments/staging/main.tf

# import vpc module  
module "webserver\_cluster" {  
 source = "../../../modules/vpc"  
 vpc\_name = "vpc-webserver  
 # ... list of variable  
}resource "aws\_instance" "webserver" {  
 count = var.webserver\_amount   
 ami = "ami-04190fdde7e961689"  
 instance\_type = var.instance\_type  
 subnet\_id = webserver\_cluster.subnet\_id  
}  
# others configuration  
...........

Pros

* Provides a straightforward approach by organizing your Terraform configurations into separate directories for each environment.
* Offers better isolation of resources and variables, as each environment has its own separate configuration files.

Cons

* Requires additional effort to manage separate state files and backends for each environment.
* Might lead to code duplication if there are common resources or modules across environments.

**3. Third-party tools:**

When managing multiple environments using Terraform, there are external tools available that can enhance your workflow and provide additional functionalities. Two popular tools in this context are [Terragrunt](https://terragrunt.gruntwork.io/" \t "_blank) and [Atlantis](https://www.runatlantis.io/).

[Terragrunt](https://terragrunt.gruntwork.io/) is a thin wrapper for Terraform that provides extra features and simplifies the management of multiple Terraform modules or configurations. It allows you to define reusable configurations, manage remote state, and handle dependencies between modules. With Terragrunt, you can easily organize your infrastructure code and avoid duplicating code across different environments.

The Power of Terragrunt:

Terragrunt is a powerful open-source tool that acts as a thin wrapper around Terraform. It provides additional features and simplifies the management of multiple Terraform modules or configurations. Let’s delve into some key benefits of using Terragrunt for managing multiple environments.

* Code Reusability
* Remote State Management
* Dependency Management
* Environment-Specific Configurations
* Terragrunt Hooks

**Scenario Approaches:**

**1. Terraform Workspaces:**

* Suitable for smaller projects with similar configurations across environments, such as development, staging, and production.
* When you want a simple and built-in way to switch between environments using the same Terraform codebase.
* When you need to isolate resources and variables specific to each environment within a shared backend state file.

**2. Environment-specific directories:**

* Useful when managing complex or diverse environments with significantly different configurations.
* When you want better control and flexibility over each environment’s resources and variables.
* When you need to manage separate state files and backends for each environment.

**3. External tools:**

* When managing larger projects with complex environments and dependencies.
* When you need advanced features like automatic environment creation, dependency management, or remote execution of Terraform commands.
* When you want to enhance automation capabilities and streamline environment management.

It’s important to consider the specific needs and constraints of your project, including the size of your infrastructure, team collaboration requirements, desired level of automation, and complexity of the environments. Evaluating these factors will help you choose the most appropriate approach for managing multiple environments with Terraform.

**PRACTICAL EXAMPLE**

**Task:** Manage AWS EC2 Instance for multiple environments.

To demonstrate how to manage multiple environments using Terraform, we will create an example of provisioning an AWS EC2 instance for both the production (prod) and staging environments.

We will explore two practical examples of managing multiple environments:

* Using Terraform workspaces
* Organizing environment-specific directories.

These approaches provide flexibility and maintainability when provisioning infrastructure across different environments.

I have provided a code repository with the relevant code examples. You can find the code for managing multiple environments with Terraform workspaces and the S3 backend in the “multiple\_environments” directory of the repository.

> git clone https://github.com/b0ld8/tf-awesome.git  
> cd multiple\_environments/  
> ls  
Mode LastWriteTime Length Name  
---- ------------- ------ ----  
d----- 7/23/2023 10:28 PM separated\_directories  
d----- 7/22/2023 8:30 PM workspaces

**1. Terraform workspaces:**

To begin, configure the backend in your Terraform configuration file ([provider.tf](http://provider.tf/)). Specify the backend block and set the necessary configuration options for using Amazon S3 as the backend. For example:

# provider.tf  
terraform {  
 required\_providers {  
 aws = {  
 source = "hashicorp/aws"  
 version = "~> 3.0"  
 }  
 }  
   
 backend "s3" {  
 bucket = "tf-awesome-backend"  
 key = "terraform.tfstate"  
 workspace\_key\_prefix = "workspaces"  
 region = "ap-southeast-1"  
 profile = "tf-awesome"  
 }  
}

We set the workspace\_key\_prefix to "workspaces/". This means that for each workspace, Terraform will create a separate directory within the S3 bucket, prefixed with "workspaces/". For instance, the state file for the "prod" workspace will be stored in "s3://tf-awesome-backend/workspaces/prod/terraform.tfstate".

After updating the backend configuration, run terraform init to initialize the backend with the new configuration.

> terraform init

The default workspace is created automatically when you initialize your Terraform project. It serves as the initial workspace where you can define and configure your infrastructure resources.

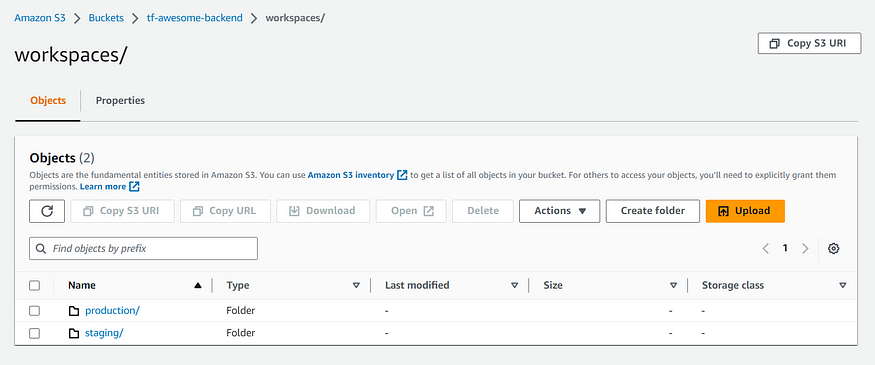
> terraform workspace list  
\* default

Create a new workspace for staging, production by running the following commands:

> terraform workspace new staging  
> terraform workspace new production  
> terraform workspace select production  
> terraform workspace list  
 default  
\* production  
 staging

Now we have two new workspaces, “staging” and “production,” created using the Terraform CLI. Each workspace will have its own separate state file and can be used to manage the infrastructure specific to that environment.

After running these commands, Terraform will manage the workspaces locally on your machine. However, if you have configured a backend like S3, Terraform will automatically create corresponding workspaces on the backend with the predefined prefix.



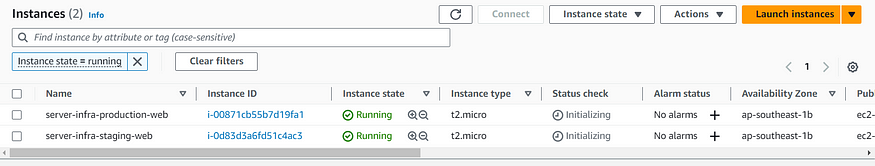
This command will execute the Terraform configuration and create the desired resources, an EC2 instance, in the production environment.

> terraform workspace select production  
> terraform plan   
> terraform apply

Repeat the steps above for the staging environment.

> terraform workspace select staging  
> terraform plan   
> terraform apply

By following these steps, you will have two EC2 instances with different names corresponding to each workspace. For example, in the production workspace, you might have an EC2 instance named “server-infra-production-web,” while in the staging workspace, you might have an EC2 instance named “server-infra-staging-web.” This separation allows you to manage and deploy resources independently in each environment.



In conclusion, Terraform workspaces offer a powerful solution for managing multiple environments. They enable separation, simplicity, collaboration, testing, versioning, and auditing, providing a robust framework for deploying and managing infrastructure across different stages of development.

**2. Environment-specific directories:**

In both of the solutions outlined, we leverage the use of modules within our projects for added convenience. These modules can be stored in separate GIT repositories, allowing for versioning and easy access. Each module is then called upon within its respective layer, and the Terraform remote states are stored in a versioned S3 backend.

The hierarchy satisfy the following requirements:

* Create a separate directory for your EC2 module, which contains the necessary Terraform configuration files for creating an EC2 instance. This module can be reused across different environments.
* In each environment directory, initialize a [provider.tf](http://provider.tf/) file to define your cloud provider configuration. Then, create a [main.tf](http://main.tf/) file where you will use the EC2 module to create an EC2 instance. Additionally, create a [variables.tf](http://variables.tf/) file to define any environment-specific variables.
* Use the EC2 module in the [main.tf](http://main.tf/) file.

/  
├── environments/  
│ ├── staging/  
│ │ ├── provider.tf  
│ │ ├── variables.tf  
│ │ └── main.tf  
│ └── prod/  
│ ├── providers.tf  
│ ├── variables.tf  
│ └── app.tf  
└── modules/  
 ├── ec2/  
 ├── main.tf  
 └── variables.tf

For each environment, such as “staging” and “production.” These directories will contain the specific Terraform configuration files for each environment.

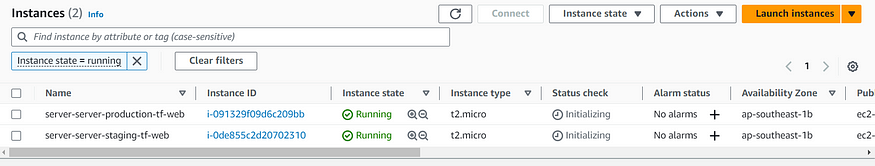
# environments/prod/provider.tf   
backend "s3" {  
 bucket = "tf-awesome-backend"  
 key = "multi-environments/prod/terraform.tfstate"  
 region = "ap-southeast-1"  
 profile = "tf-awesome"  
 }  
......  
# environments/staging/provider.tf   
backend "s3" {  
 bucket = "tf-awesome-backend"  
 key = "multi-environments/staging/terraform.tfstate"  
 region = "ap-southeast-1"  
 profile = "tf-awesome"  
 }  
....

Access the specific environment directory: From the command line, access the directory of the environment you want to deploy. For example, navigate to the “staging” directory if you want to deploy in the staging environment.

cd ./prod   
cd ./staging

Deploy the infrastructure: Run the Terraform commands, such as terraform init to initialize the environment, terraform plan to review the changes, and terraform apply to deploy the infrastructure specific to that environment.

> terraform init  
> terraform plan   
> terraform apply



In summary, using separate directories with Terraform modules for managing multiple environments enables reusability, environment-specific configurations, clear separation, easy rollbacks, improved collaboration, and efficient deployment. This approach provides a structured and organized way to manage infrastructure across different environments, ensuring consistency and reducing the likelihood of errors or conflicts.

**Conclusion**

Terraform workspaces offer a powerful solution for managing multiple environments. They enable separation, simplicity, collaboration, testing, versioning, and auditing, providing a robust framework for deploying and managing infrastructure across different stages of development. By following the best practices outlined in this blog post, you can effectively manage and deploy infrastructure in a multi-environment setup using Terraform.