

## MODULE IN TERRAFORM

In an organization, multiple teams (Developer Team, Testing Team, Production Team) are collaborating on a project that involves working with a Cloud provider using Terraform. Among the tasks assigned to the teams is the launch of an EC2 Instance from AWS Cloud.

There are two approaches to accomplish this task:

1. Each team can individually write the code for launching the EC2 Instance.
2. Alternatively, the teams can opt for a more centralized approach. They can create a common repository or location where they store the code for launching the EC2 Instance. This central repository would contain Terraform modules that encapsulate the necessary configuration for launching an EC2 Instance. All teams can then utilize these modules to launch the EC2 Instance consistently, eliminating redundant efforts and promoting standardization across the organization. This way, the teams can collaboratively maintain and enhance the codebase while leveraging the power of Terraform modules to streamline the process.

```
Himanshu Kumar@AICPL-D010 MINGW64 ~/Documents/Terraform/Day 3
$ mkdir Team-Dev Team-Test Team-Prod Modules

Himanshu Kumar@AICPL-D010 MINGW64 ~/Documents/Terraform/Day 3
$ ls
Modules/ Team-Dev/ Team-Prod/ Team-Test/ terraform.exe*
```

I have created four directories: "Team-Dev," "Team-Test," "Team-Prod," and "Modules." The "Modules" directory is designated to store various modules related to resources such as EC2 and EBS. The other three directories, "Team-Dev," "Team-Test," and "Team-Prod," contain specific work related to their respective teams.

In "Team-Dev", Here I am Creating a File with the name of Provider.tf.

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

provider "aws" {
  region = "ap-south-1"
  profile = "default"
}
```

In this configuration:

- The terraform block indicates the required provider for the Terraform project.
- The required\_providers block specifies that the project requires the "aws" provider.
- The aws provider is sourced from the official Terraform registry, provided by HashiCorp.
- The provider version constraint ~> 5.0 means that any version within the 5.x range is acceptable, with the latest patch version being used.
- The provider block configures the "aws" provider with the region set to "ap-south-1" (Asia Pacific, Mumbai) and the AWS profile set to "default," which typically refers to the default AWS credentials and settings on the system where Terraform is being executed.

I have created a centralized module for EC2 Instances inside the "modules" directory. Here's the code written in the module.

#### # Centralized Module For EC2 Instance

```
resource "aws_instance" "web" {
  ami = "ami-0ded8326293d3201b"
  instance_type = "t2.micro"
  tags = {
    Name = "AWS OS"
  }
}
```

This Terraform module defines an AWS EC2 Instance resource named "web." It specifies the Amazon Machine Image (AMI) as "ami-0ded8326293d3201b," the instance type as "t2.micro," and assigns a tag with the name "AWS OS" to the EC2 Instance. By using this centralized module, multiple teams can now easily launch EC2 Instances with consistent configurations and tags, promoting standardization and reducing duplication of code across the organization.

In the "Main.tf" file created inside the "Team-Dev" directory, I have specified the following code, utilizing a Terraform module named "EC2-Module" with a relative path to the "EC2" module directory:

```
module "EC2-Module" {
  source = "../Modules/EC2"
}
```

In this configuration:

- The module block defines a module named "EC2-Module."
- The source attribute specifies the relative path to the "EC2" module directory, which is located one level up from the "Team-Dev" directory (indicated by ..).
- By utilizing this module block with the specified source, the Terraform configuration inside the "EC2" module directory will be imported and used within the context of the "Team-Dev" configuration. This enables reusability and modularity in managing EC2 instances across different teams within the organization.

Now I am going to run the "terraform init" command

```
PS C:\Users\Himanshu Kumar\Documents\Terraform\Day 3\Team-Dev> .\terraform.exe init
```

Initializing the backend...

Initializing modules...

Initializing provider plugins...

- Finding hashicorp/aws versions matching "~> 5.0"...
- Installing hashicorp/aws v5.10.0...
- Installed hashicorp/aws v5.10.0 (signed by HashiCorp)

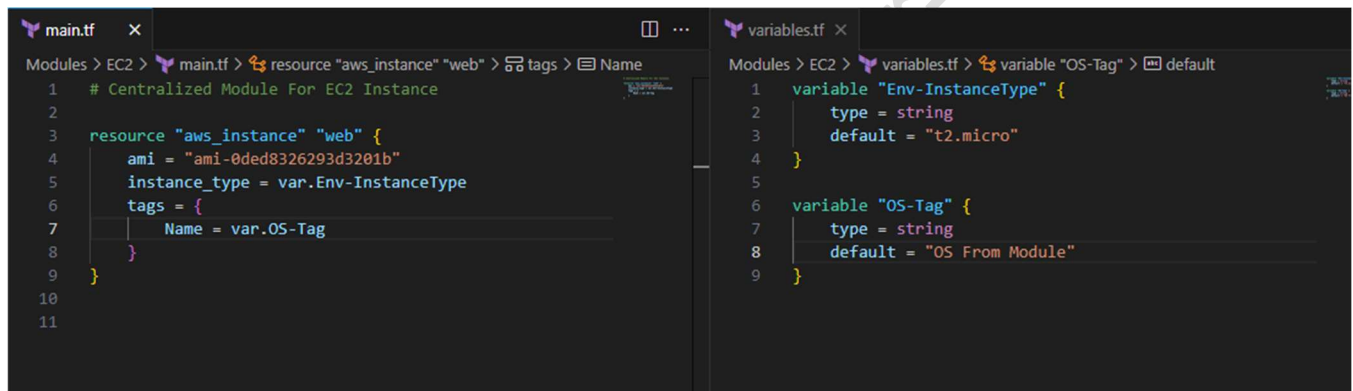
Terraform has created a lock file **.terraform.lock.hcl** to record the provider selections it made above. Include this file in your version control repository so that Terraform can guarantee to make the same selections by default when you run "terraform init" in the future.

**Terraform has been successfully initialized!**

Sometimes, we encounter scenarios where we need to use modules, but within those modules, we want to customize certain inputs. In the previous example, the **OS Name, AMI-ID, and Instance Type were fixed**. However, there might be situations where we want to **provide different values for these fields**.

To achieve this, we have several options:

1. **Use Variables Inside Modules:** We can define variables inside the module to allow users to provide custom inputs. By using variables, users can pass values to the module during its usage and tailor the resources accordingly.
2. **Utilize terraform.tfvars Files:** Another approach is to create a separate `terraform.tfvars` file to store input values for the module. This file contains variable assignments that override default values. Terraform automatically loads this file if it exists, simplifying the process of passing custom inputs.
3. **Accept Input Directly from Users (Not Recommended):** Though possible, directly taking input from users during runtime is generally discouraged. It can lead to unintended mistakes and make the Terraform workflow less predictable and less reproducible.
4. **Module Blocks:** Alternatively, we can use module blocks directly in the configuration where we call the module. Module blocks allow us to provide input values directly as arguments within the module call, overriding the default values defined in the module.



The screenshot shows two Terraform files in a code editor. The left pane shows `main.tf` with a resource definition for an AWS EC2 instance. The right pane shows `variables.tf` with two variable definitions: `Env-InstanceType` and `OS-Tag`.

```
main.tf
1 # Centralized Module For EC2 Instance
2
3 resource "aws_instance" "web" {
4     ami = "ami-0ded8326293d3201b"
5     instance_type = var.Env-InstanceType
6     tags = {
7         Name = var.OS-Tag
8     }
9 }
10
11

variables.tf
1 variable "Env-InstanceType" {
2     type = string
3     default = "t2.micro"
4 }
5
6 variable "OS-Tag" {
7     type = string
8     default = "OS From Module"
9 }
```

In this configuration:

**main.tf** contains the centralized module definition for the AWS EC2 Instance. The `aws_instance` resource creates an EC2 Instance with a specific AMI (Amazon Machine Image) defined by the AMI ID "ami-0ded8326293d3201b". The `instance_type` parameter is set to the value of the `Env_InstanceType` variable, allowing users to customize the instance type when using this module. The `tags` parameter assigns a `Name` tag to the EC2 Instance with the value provided in the `OS_Tag` variable.

**variables.tf** contains the variable definitions. Two variables are defined: `Env_InstanceType` and `OS_Tag`. The `Env_InstanceType` variable is of type string with a default value of "t2.micro". This default value will be used if no other value is provided when using the module. Similarly, the `OS_Tag` variable is also of type string with a default value of "OS From Module".

By using variables, users can now easily customize the instance type and instance tag when using the centralized module in different parts of their Terraform configuration.

When using modules to launch instances, we might want to monitor outputs like the public IP or attached storage of the operating system. However, accessing these outputs directly from the module can be difficult. To overcome this, we can define output fields within the module, allowing us to export specific information from the module to the calling code. This way, we can easily import and utilize these outputs in other parts of our Terraform configuration, making it more convenient to inspect the critical information related to the launched instances.

```
main.tf
Team-Dev > main.tf > ...
9 }
10
11 # Importing Output From Modules -> EC2
12 output "Team-Dev_EC2_Module_Output_Import" {
13   value = module.EC2-Module.EC2-Module_Output
14 }
15
16 output "Team-Dev_PublicIP" {
17   value = module.EC2-Module.Default-Public_IP
18 }
19

output.tf
Modules > EC2 > output.tf > output "Default-Public_IP"
1 # Output Export To Module
2 output "EC2-Module_Output" {
3   value = "Output From EC2 Module"
4 }
5
6 output "Default-Public_IP" {
7   value = aws_instance.web.public_ip
8 }
9
```

In this calling code, three outputs are defined:

**Team-Dev:** This output provides a static value "Team Dev is Start working" as a result.

**Team-Dev\_EC2\_Module\_Output\_Import:** This output imports the value of the EC2-Module\_Output output from the "EC2" module and displays it here.

**Team-Dev\_PublicIP:** This output imports the value of the Default-Public\_IP output from the "EC2" module and displays the public IP address of the EC2 instance created within the module.

By defining outputs in the module and calling them from other parts of the configuration, you can export information from modules and use it in different parts of your Terraform code, enhancing the modularity and reusability of your infrastructure configuration.

In this module, two outputs are defined:

**EC2-Module\_Output:** This output provides a static value "Output From EC2 Module" as a result.

**Default-Public\_IP:** This output uses the aws\_instance.web.public\_ip attribute to export the public IP address of the EC2 instance created within the module.

While creating Modules always create 3 Files

- Main.tf
- Variable.tf
- Output.tf

## Data Source

In Terraform, a data source is a way to retrieve external data that already exists outside of the configuration. It allows you to fetch information from various sources, such as cloud providers, APIs, or even local files, and use that data in your Terraform code. Data sources are read-only, meaning they do not create or modify resources like resource blocks do; they only provide information to be used elsewhere in the configuration.

For example, in the context of retrieving data from an Amazon Machine Image (AMI) in AWS, you can use the "aws\_ami" data source. The "aws\_ami" data source allows you to fetch information about an existing AMI by specifying its ID, name, or other attributes. This data source is useful when you want to use an existing AMI as a basis for creating new instances or other resources in your infrastructure.

The provided Terraform code defines a centralized module for an EC2 Instance using a data source to retrieve the AMI ID dynamically from AWS. Let's go through the code to understand it better:

```
# Centralized Module For EC2 Instance

resource "aws_instance" "web" {
  ami = data.aws_ami.FetchAMI.id
  instance_type = var.Env-InstanceType
  tags = {
    Name = var.OS-Tag
  }
}
```

In this part of the code:

- The aws\_instance resource block creates an EC2 Instance named "web."
- The ami attribute specifies the AMI ID using the data.aws\_ami.FetchAMI reference. This means the actual AMI ID will be fetched dynamically from the data source named "FetchAMI."
- The instance\_type attribute is set to the value of the variable var.Env-InstanceType. It indicates the instance type to be used for the EC2 Instance. The actual value will be provided when using the module.
- The tags attribute is used to assign tags to the EC2 Instance. In this case, the Name tag is set to the value of the variable var.OS-Tag.

```
# Retrive AMI ID
data "aws_ami" "FetchAMI" {
  most_recent      = true
  owners           = ["amazon"]
  filter {
    name   = "name"
    values = ["amzn2-ami-*-gp2"]
  }
  filter {
    name   = "root-device-type"
    values = ["ebs"]
  }

  filter {
    name   = "virtualization-type"
    values = ["hvm"]
  }
}
```

```

filter {
  name = "architecture"
  values = ["x86_64"]
}
}

```

In this part of the code:

- The data block declares a data source named "aws\_ami" with the alias "FetchAMI" for reference.
- The data source fetches the most recent Amazon Linux 2 (amzn2) AMI that uses the "gp2" (General Purpose SSD) root device type, HVM virtualization type, and x86\_64 architecture.
- The most\_recent attribute is set to true to ensure the data source fetches the most recent AMI.
- The owners attribute specifies that the AMI must be owned by the "amazon" account.
- The filter blocks are used to define specific criteria for selecting the desired AMI.

With this configuration, when you use the "aws\_instance" resource from the centralized module, it will dynamically fetch the most recent AMI ID that matches the specified criteria, making your Terraform configuration more flexible and automated.

```

main.tf
Team-Dev > main.tf > ...
11 # Importing Output From Modules -> EC2
12 output "Team-Dev_EC2_Module_Output_Import" {
13   value = module.EC2-Module.EC2-Module_Output
14 }
15
16 output "Team-Dev_PublicIP" {
17   value = module.EC2-Module.Default-Public_IP
18 }
19
20 output "Team-Dev_AMIID" {
21   value = module.EC2-Module.AMI-ID
22 }

output.tf
Modules > EC2 > output.tf > ...
1 # Output Export To Module
2 output "EC2-Module_Output" {
3   value = "Output From EC2 Module"
4 }
5
6 output "Default-Public_IP" {
7   value = aws_instance.web.public_ip
8 }
9
10 output "AMI-ID" {
11   value = "AMI ID is: ${aws_instance.web.ami}"
12 }
13

```

### Outputs:

```

Team-Dev = "Team Dev is Start working"
Team-Dev_AMIID = "AMI ID is: ami-086854ff7f3ebd39b"
Team-Dev_EC2_Module_Output_Import = "Output From EC2 Module"
Team-Dev_PublicIP = "15.206.74.128"

```

As we can see in the output: AMI ID is retrieved from the AMAZON.



### # terraform fmt

This command is used to automatically format Terraform configuration files, making them consistent and adhering to the Terraform style conventions. It ensures proper indentation, line breaks, and overall code structure. Running ``terraform fmt`` is a good practice to maintain readable and organized Terraform code.

### # terraform validate

This command is used to validate the syntax and configuration of Terraform files in the current working directory. It checks for errors and potential issues in the configuration files without actually executing the infrastructure changes.

When you run `terraform validate`, Terraform will parse and validate the Terraform files (usually with a `.tf` extension) and will provide feedback if there are any syntax errors or other problems that prevent the configuration from being correctly processed.

### # terraform plan

The ``terraform plan`` command is used to create an execution plan for Terraform. When you run this command, terraform will analyze the current state of your infrastructure and compare it with the configuration in your Terraform files (usually with a `.tf` extension).

The ``terraform plan`` process performs the following steps:

1. Reads the configuration files: Terraform reads the `.tf` files in the current directory to understand the desired state of your infrastructure.
2. Initializes the backend: If not already initialized, terraform will also initialize the backend to store the state file.
3. Downloads provider plugins: Terraform downloads the necessary provider plugins to interact with the cloud providers or services mentioned in the configuration.
4. Analyzes the state: Terraform inspects the current state of your infrastructure stored in the state file to understand what resources are already provisioned.
5. Creates a plan: After comparing the desired state from the configuration with the current state, Terraform creates a plan detailing what changes it will make to achieve the desired state. This includes creating, updating, or deleting resources as needed.
6. Outputs the plan: The execution plan is then presented in human-readable format, showing the actions Terraform will take to apply the changes.

The ``terraform plan`` command is useful for previewing the changes Terraform will make before actually applying them with ``terraform apply``. It helps in understanding what resources will be affected and provides an opportunity to review the changes before applying them to your infrastructure.

### # terrafrom apply

The ``terraform apply`` command is used to apply the changes defined in your Terraform configuration to your infrastructure. When you run this command, Terraform will execute the planned changes that were generated by the ``terraform plan`` command.

Here's what happens when you run ``terraform apply``:

1. Reads the configuration files: Terraform reads the `.tf` files in the current directory to understand the desired state of your infrastructure.
2. Initializes the backend: If not already initialized, Terraform will also initialize the backend to store the state file.
3. Downloads provider plugins: Terraform downloads the necessary provider plugins to interact with the cloud providers or services mentioned in the configuration.
4. Execution plan: If you have previously run `terraform plan`, the plan generated during that step is now applied. If not, Terraform will create a new execution plan before applying the changes.
5. Confirmation: Terraform will present a summary of the changes it plans to make and will prompt you to confirm whether you want to proceed with applying those changes. You must explicitly type "yes" to proceed with the changes.
6. Applies changes: After confirmation, Terraform begins to apply the changes, creating, updating, or deleting resources as specified in the Terraform configuration.
7. State file update: As Terraform applies the changes, it updates the state file to reflect the current state of your infrastructure. The state file is crucial for tracking and managing the resources Terraform provisions.
8. Outputs: After the changes are applied, Terraform will display any outputs specified in your configuration. These outputs can be useful information about the resources that were created or modified.

It's essential to review the plan generated by `terraform plan` carefully before executing `terraform apply`. This helps to avoid unintentional changes to your infrastructure. Additionally, ensure that you have a backup of your Terraform state file and understand the implications of the changes being applied.