Terraform modules

Even when you don't create a module intentionally, if you use Terraform, you are already writing a module – a so-called "root" module.

Any Terraform configuration file (.tf) in a directory, even just one, forms a module.

What does a module do?

A Terraform module allows you to create logical abstraction on the top of some resource set. In other words, a module allows you to group resources together and reuse this group later, possibly many times.

Let's assume we have a virtual server with some features hosted in the cloud. What set of resources might describe that server? For example:

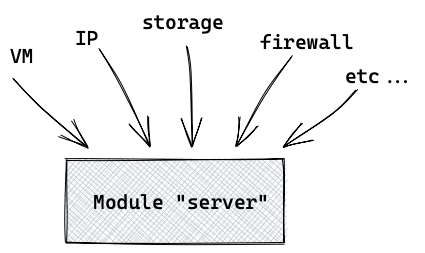
the virtual machine itself, created from some image

an attached block device of a specified size for additional storage

a static public IP mapped to the server's virtual network interface

a set of firewall rules to be attached to the server

other things like another block device, additional network interface, and so on



Now let's assume that you need to create this server with a set of resources many times. This is where modules are really helpful – you don't want to repeat the same configuration code over and over again, do you?

Here is an example that illustrates how our "server" module might be called.  
"To call a module" means to use it in the configuration file.

Here we create 5 instances of the "server" using single set of configurations (in the module):

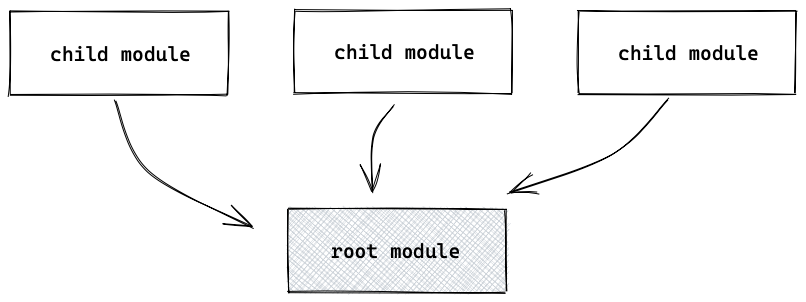
module "server" {

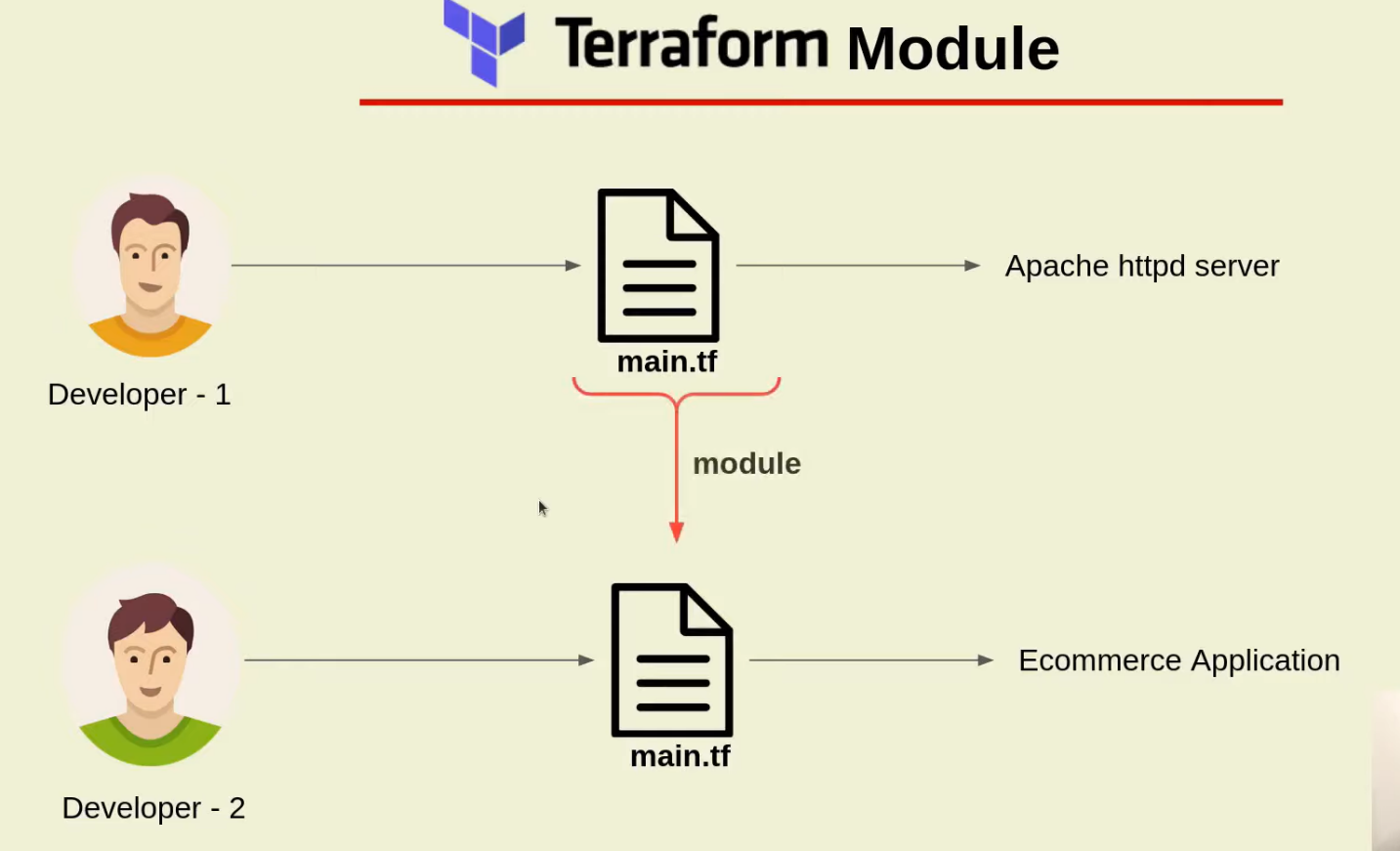
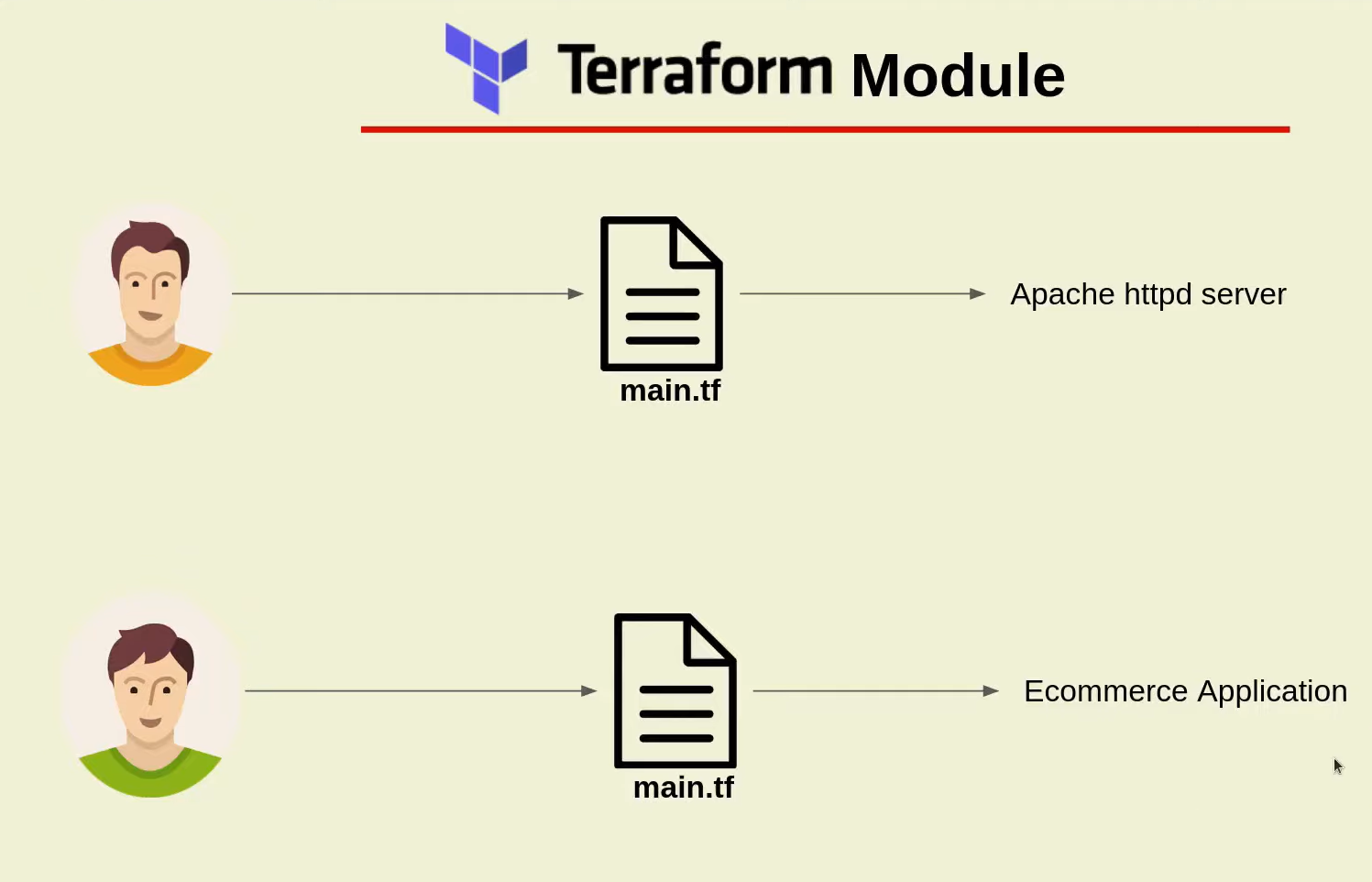
count = 5

source = "./module\_server"

some\_variable = some\_value

}





How Terraform modules works?

[Terraform](https://www.terraform.io/) is an IaC(infrastructure as Code) framework for managing and provisioning the infrastructure but have you ever thought of creating only a single terraform configuration file for managing the complete cloud infrastructure.

Well, it sounds insane because if you only have a single file for managing the complete infrastructure then it will grow in complexity as well the line of configuration code will be manifold. That is why terraform introduced a concept of ***module*** which will help you to organize your terraform configuration so that you can re-use the configuration and keep your terraform code more clean and modular.

In this blog post, we will create two modules *module-1* and *module-2*. Inside each module, we will install the [apache httpd server](https://httpd.apache.org/" \t "_blank) and nginx server with each module is having its home page.

Steps for creating terraform modules along with best practices and benefits

[Create your first module](https://jhooq.com/terraform-module/#1-create-a-your-first-module)

[Module structure](https://jhooq.com/terraform-module/#2-module-structure)

[Calling the module](https://jhooq.com/terraform-module/#3-calling-the-module)

[Module Inputs](https://jhooq.com/terraform-module/#4-module-inputs)

[Module Output](https://jhooq.com/terraform-module/#5-module-output)

[Module best practices](https://jhooq.com/terraform-module/#4-module-best-practices)

[Benefits of Modules](https://jhooq.com/terraform-module/#5-benefits-of-modules#5-benefits-of-modules)

1. Create your first module

To keep things simple we will be creating two modules named ***module-1, module-2***.

Each module will have its own main.tf files.

Terraform modules with its own main.tf files

1.1Install apache httpd server and Nginx in module-1 and module-2

To keep things more regular we will install two [apache httpd servers](https://httpd.apache.org/" \t "_blank) in each module.

Let us start with the ***module-1*** -

**1. Specify the terraform required version**

terraform {

required\_version = ">=0.12"

}

**2. Create an ec2***aws\_instance***along with the***user\_data***block in which we will be writing bash commands for installing the apache2 httpd server.**

*(Note - Here in this instance I have used the key\_name aws\_key. Follow this post on*[*how to create aws keys for your ec2 instance*](https://jhooq.com/terraform-ssh-into-aws-ec2/)*)*

resource "aws\_instance" "ec2\_example" {

ami = "ami-0767046d1677be5a0"

instance\_type = "t3.micro"

key\_name= "mr-cloud-book"

vpc\_security\_group\_ids = [aws\_security\_group.main.id]

}

and here is the *user\_data* block for installing the apach2 httpd server.

We will be using the same *user\_data* block for installing the apache2 httpd server .

user\_data = <<-EOF

#!/bin/bash

sudo su

yum update -y

yum install -y httpd

cd /var/www/html

wget https://github.com/azeezsalu/techmax/archive/refs/heads/main.zip

unzip main.zip

cp -r techmax-main/\* /var/www/html/

rm -rf techmax-main main.zip

systemctl enable httpd

systemctl start httpd

EOF

Module – 2

user\_data = <<-EOF #cd /etc/nginx #cat nginx.conf

#!/bin/sh

sudo yum install epel-release #cd /usr/share/nginx/html

sudo yum update

sudo amazon-linux-extras install nginx1 –y

sudo systemctl status nginx

sudo systemctl start nginx

EOF

**3. Define the***aws\_security\_group***along with ingress rules for port***80***and***22***.**

We need to open the *port 80* on the ec2 instance to access the apache httpd server home page and *port 22* will need to ssh into the ec2 instance.

Here is the resource block -

resource "aws\_security\_group" "main" {

name = "EC2-webserver-SG-2"

description = "Webserver for EC2 Instances"

ingress {

from\_port = 80

protocol = "TCP"

to\_port = 80

cidr\_blocks = ["0.0.0.0/0"]

}

ingress {

from\_port = 22

protocol = "TCP"

to\_port = 22

cidr\_blocks = ["115.97.103.44/32"]

}

egress {

from\_port = 0

protocol = "-1"

to\_port = 0

cidr\_blocks = ["0.0.0.0/0"]

}

}

*BASH*

**4. Complete terraform script for module-1 and module -2**

***module-1***

terraform {

required\_version = ">=0.12"

}

resource "aws\_instance" "ec2\_example" {

ami = "ami-0767046d1677be5a0"

instance\_type = "t3.micro"

key\_name= "mr-cloud-book"

vpc\_security\_group\_ids = [aws\_security\_group.main.id]

user\_data = <<-EOF

#!/bin/bash

sudo su

yum update -y

yum install -y httpd

cd /var/www/html

wget https://github.com/azeezsalu/techmax/archive/refs/heads/main.zip

unzip main.zip

cp -r techmax-main/\* /var/www/html/

rm -rf techmax-main main.zip

systemctl enable httpd

systemctl start httpd

EOF

}

resource "aws\_security\_group" "main" {

name = "EC2-webserver-SG-1"

description = "Webserver for EC2 Instances"

ingress {

from\_port = 80

protocol = "TCP"

to\_port = 80

cidr\_blocks = ["0.0.0.0/0"]

}

ingress {

from\_port = 22

protocol = "TCP"

to\_port = 22

cidr\_blocks = ["0.0.0.0/0"]

}

egress {

from\_port = 0

protocol = "-1"

to\_port = 0

cidr\_blocks = ["0.0.0.0/0"]

}

}

***module-2***

terraform {

required\_version = ">=0.12"

}

resource "aws\_instance" "ec2\_example" {

ami = "ami-0767046d1677be5a0"

instance\_type = "t2.micro"

key\_name= "aws\_key"

vpc\_security\_group\_ids = [aws\_security\_group.main.id]

user\_data = <<-EOF

#!/bin/bash

      sudo su

      yum update -y

      amazon-linux-extras install nginx1 -y

      systemctl enable nginx

      systemctl start nginx

      systemctl status nginx

      sudo echo <!DOCTYPE html> <html> <head> <meta name="viewport" content="width=device-width, initial-scale=1"> <title>youtube Allow Fullscreen</title> </head> <body> <!--Need Internet Connection--> <!--Fullscreen allow--> <iframe width="420" height="315" src="https://www.youtube.com/embed/OK7fy40Ai6A" allowfullscreen></iframe> </body> </html>" > /usr/share/nginx/html/index.html

      systemctl restart nginx

}

resource "aws\_security\_group" "main" {

name = "EC2-webserver-SG-1"

description = "Webserver for EC2 Instances"

ingress {

from\_port = 8080

protocol = "TCP"

to\_port = 8080

cidr\_blocks = ["0.0.0.0/0"]

}

ingress {

from\_port = 22

protocol = "TCP"

to\_port = 22

cidr\_blocks = ["115.97.103.44/32"]

}

egress {

from\_port = 0

protocol = "-1"

to\_port = 0

cidr\_blocks = ["0.0.0.0/0"]

}

}

...

*BASH*

2. Module structure

In the [step 1](https://jhooq.com/terraform-module/#1-create-a-your-first-module) we have seen the main.tf terraform files of module-1 and module-2.

Now let's talk about the complete structure of your terraform project where you will have your parent main.tf file which will be calling the module-1 and module-2

terraform modules with parent module terraform file main.tf

3. Calling the module

Alright in the previous steps we have seen how our modules main.tf files along with the structure of the modules along with the parent main.tf files.

Once you define the modules the next important step would be to call the modules from the parent main.tf file and terraform makes it easy to call the modules based on the relative paths-

Here is my parent main.tf calling the *module-1* and *module-2*

provider "aws" {

region = var.web\_region

access\_key = var.access\_key

secret\_key = var.secret\_key

}

module "webserver-1" {

source = ".//module-1"

}

module "webserver-2" {

source = ".//module-2"

}

*BASH*

4. Module Inputs

Just like any other programming language terraform also supports re-usability with the help of terraform module. You might be familiar with the concept of functions in other programming languages. In general functions always have some input parameter which needs to passed during the function call, similarly terraform module can also accept input parameters.

**Example -**

Let's take the same example where we have defined the **module-1** but instead of **hard coding ec2 instance type** let's create an input variable which can be passed later -

1. Create a input variable inside module

Here is code snippet of **module-1** where I have replaced hardcoded **instance type** to a variable var.web\_instance\_type -

***Path****- terraform-modules/module-1/main.tf*

resource "aws\_instance" "ec2\_module\_1" {

ami = var.ami\_id

instance\_type = var.web\_instance\_type

key\_name= "aws\_key"

vpc\_security\_group\_ids = [aws\_security\_group.main.id]

user\_data = <<-EOF

#!/bin/sh

sudo apt-get update

sudo apt install -y apache2

sudo systemctl status apache2

sudo systemctl start apache2

sudo chown -R $USER:$USER /var/www/html

sudo echo "<html><body><h1>Hello this is module-1 at instance id `curl http://169.254.169.254/latest/meta-data/instance-id` </h1></body></html>" > /var/www/html/index.html

EOF

}

*BASH*

2. Pass the module input variable value while calling terraform module

In the following code snippet we are calling **module-1** but also we are passing input variable .i.e. *web\_instance\_type="t2.large"*, so based on your need and environment you can update instance type without changing the code your terraform module.

***Path****- terraform-modules/main.tf*

provider "aws" {

region = var.web\_region

access\_key = var.access\_key

secret\_key = var.secret\_key

}

module "webserver-1" {

source = ".//module-1"

web\_instance\_type = "t2.large"

}

module "webserver-2" {

source = ".//module-2"

}

*BASH*

5. Module Output

Terraform module also have the capability to produce the return output just like function which do return some value back after calling them.

**Example -**

1. Create an output variable public\_ip\_ec2

Let's again take the example of same terraform **module-1**, define an output block inside terraform **module-1**.

The following code snippet is for adding an **output variable** inside **module-1** *(*[*Refer to GitHub for code repo*](https://github.com/rahulwagh/Terraform-Topics.git)*)*

***Path****- terraform-modules/module-1/main.tf*

resource "aws\_instance" "ec2\_module\_1" {

ami = var.ami\_id

instance\_type = var.web\_instance\_type

key\_name= "aws\_key"

vpc\_security\_group\_ids = [aws\_security\_group.main.id]

user\_data = <<-EOF

#!/bin/sh

sudo apt-get update

sudo apt install -y apache2

sudo systemctl status apache2

sudo systemctl start apache2

sudo chown -R $USER:$USER /var/www/html

sudo echo "<html><body><h1>Hello this is module-1 at instance id `curl http://169.254.169.254/latest/meta-data/instance-id` </h1></body></html>" > /var/www/html/index.html

EOF

}

output "public\_ip\_ec2" {

value = aws\_instance.app\_server.public\_ip

description = "Public IP address of EC2 instance"

}

*BASH*

2. Use module output variable to show the output

After declaring module output variable inside terraform module we can refer to the output variable inside **main. tf**. But to access the output variable you need to use following syntax

module.<MODULE\_NAME>.<OUTPUT\_VARIABLE\_NAME>

*BASH*

Here is the **main.tf** accessing the output variable public\_ip\_ec2

***Path****- terraform-modules/main.tf*

provider "aws" {

region = var.web\_region

access\_key = var.access\_key

secret\_key = var.secret\_key

}

module "webserver-1" {

source = ".//module-1"

web\_instance\_type = "t2.large"

}

output "public\_ip\_ec2" {

value = module.module-1.public\_ip\_ec2

description = "Public IP of EC2"

}

module "webserver-2" {

source = ".//module-2"

}

5. Module best practices

You must always write the provider name inside your terraform file. For example, if you are using the AWS then the following line should be added as provider -

provider "aws" {

}

For [google](https://cloud.google.com/) you can write the provider in the following manner -

provider "google" {

}

While you are trying to write you're terraforming module always keep in mind that you have to go for simplicity and not to increase the complexity of your terraform project. Always keep the modules bare minimum if possible so that it will help other developers to understand and troubleshoot the issues if needed.

Terraform has a very good concept of local modules which you use to encapsulate your infrastructure logic. Use of modules is always recommended from the start of the project so that you have more control over the terraform code and it does not get spread sporadically.

Refer to the terraform public registry for finding more stable modules so that you do not have to re-invent the wheel.

Always keep and habit of publishing the modules so that it can be used by other teams members also

6. Benefits of Modules

Well, you will always get benefited from modules if you implement it properly. But here are certain advantages you will get with the terraform modules -

***Organize configuration -*** With modules it is always easy to navigate and it helps any developer to understand the terraform project with ease. Certain with the module you can break down very complex infrastructure modules into very simplistic terraform modules.

***Encapsulation -*** You can benefit from the encapsulation also and with terraform module you can hide the internal implementation of your infrastructure set up so that you can prevent unwanted changes happening to your modules by other developers.

***Re-usability -*** If you have broken down your infrastructure into smaller and generic modules then it would help you to re-use the modules into another infrastructure setup.

***Consistency -*** Following the best practices(encapsulation, organize, simple terraform module) will help you to achieve consistent behavior across all the different environments which will help you to reduce the debugging time and reduce the infrastructure maintenance cost.