Terraform Modules

Configuration files can be separated out into modules to better organize your configuration. This makes your code easier to read and reusable across your organization. You can also use the Public Module Registry to find pre-configured modules.

- Task 1: Refactor your existing code into a local module
- Task 2: Explore the Public Module Registry and install a module
- Task 3: Refresh and rerun your Terraform configuration

Task 1: Refactor your existing code into a local module

A Terraform module is just a set of configuration. For this lab, we'll refactor your existing configuration so that the webserver configuration is inside a module.

Step 6.1.1

Create a new directory called `server` in your `/workspace/terraform` directory and create a new file inside of it called `server.tf`.

Step 6.1.2

Edit the file `server/server.tf`, with the following contents:

```
variable ami {}
variable subnet_id {}
variable vpc_security_group_ids {
  type = list
}
variable identity {}

resource "aws_instance" "web" {
  ami                    = var.ami
  instance_type          = "t2.micro"
  subnet_id              = var.subnet_id
  vpc_security_group_ids = var.vpc_security_group_ids

  tags = {
    "Identity"    = var.identity
    "Name"        = "Student"
```


```
    "Environment" = "Training"
  }
}

output "public_ip" {
  value = aws_instance.web.public_ip
}

output "public_dns" {
  value = aws_instance.web.public_dns
}
```

Step 6.1.3

In your root configuration (also called your root module) `/workspace/terraform/main.tf`, we can remove the previous references to your configuration and refactor them as a module.

```
# Remove the entire block:
#
# resource "aws_instance" "web" {
# ...
# }

module "server" {
  source = "./server"

  ami                    = var.ami
  subnet_id              = var.subnet_id
  vpc_security_group_ids = var.vpc_security_group_ids
  identity               = var.identity
}
```

Our outputs should now be defined in their own file `outputs.tf` on the same file level as the root module. At the bottom of your root `output.tf` configuration, modify the public IP and public DNS outputs to match the following. Notice the difference in interpolation now that the information is being delivered by a module.

```
output "public_ip" {
  value = module.server.public_ip
}

output "public_dns" {
  value = module.server.public_dns
}
```


Step 6.1.4

Now run `terraform get` or `terraform init` to install the module. Since we're just adding a module and no providers, `get` is sufficient, but `init` is safe to use too. Even local modules need to be installed before they can be used.

Once you've done that, you can run `terraform apply` again. Notice that the the instance will be recreated, and its id changed, but everything else should remain the same.

```
terraform apply
```

```
# aws_instance.web will be destroyed
- resource "aws_instance" "web" {

...

# module.server.aws_instance.web will be created
+ resource "aws_instance" "web" {

...
```

Task 2: Explore the Public Module Registry

Step 6.2.1

HashiCorp hosts a public module registry at: https://registry.terraform.io/

The registry contains a large set of community-contributed modules that you can use in your own configurations. Explore the registry to see what is available to you.

Step 6.2.2

Search for "dynamic-keys" in the public registry and uncheck the "Verified" checkbox. You should then see a module called "dynamic-keys" created by one of HashiCorp's founders, Mitchell Hashimoto. Alternatively, you can navigate directly to https://registry.terraform.io/modules/mitchellh/dynamic-keys/aws/2.0.0.

Select this module and read the content on the Readme, Inputs, Outputs, and Resources tabs. This module will generate a public and private key pair so you can SSH into your instance.

Step 6.2.3

To integrate this module into your configuration, add this after your provider block in `main.tf`:

```
module "keypair" {
  source  = "mitchellh/dynamic-keys/aws"
  version = "2.0.0"
  path    = "${path.root}/keys"
  name    = "${var.identity}-key"
}
```

****This module exposes the private key information in the Terraform state and should not be used in production!****

Now you're referring to the module, but Terraform will need to download the module source before using it. Run the command `terraform init` to download it.

To provision the resources defined by the module, run `terraform apply`, and answer `yes` to the confirmation prompt.

Step 6.2.4

Now we'll use the *keypair* module to install a public key on our server. In `main.tf`, add the necessary output from our key module to our server module:

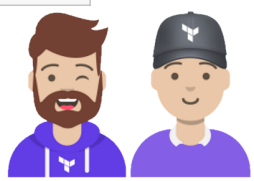
```
module "server" {
  source = "./server"

  ami                    = var.ami
  subnet_id              = var.subnet_id
  vpc_security_group_ids = var.vpc_security_group_ids
  identity               = var.identity
  key_name               = module.keypair.key_name
  private_key            = module.keypair.private_key_pem
}
```

Step 6.2.5

In your `server/server.tf` file, add two new variables to the rest of the variables at the top of the file:

```
variable key_name {}
variable private_key {}
```



Add the *key_name* variable to the *aws_instance* resource block in `server/server.tf`:

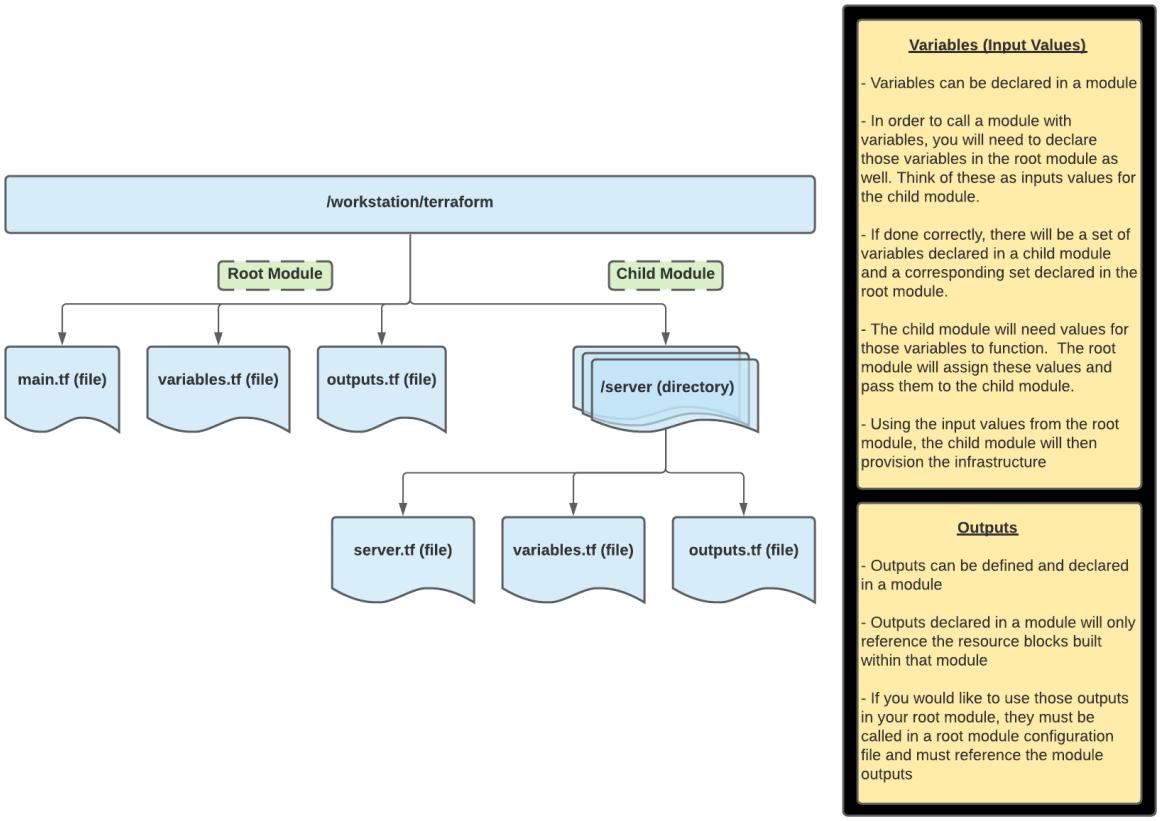
```
resource "aws_instance" "web" {
  ami                    = var.ami
  instance_type          = "t2.micro"
  subnet_id              = var.subnet_id
  vpc_security_group_ids = var.vpc_security_group_ids
  key_name               = var.key_name

  # ... leave the rest of the block unchanged
}
```

We'll use the private_key variable later.

File structure diagram

Modules can be a bit daunting at first. We are suddenly working with a number of files in different file hierarchies, and those files are referencing each other in a few different ways. This diagram is a general overview of how the files interact with each other, and how variables and outputs are defined and declared between root modules and child modules.



Task 3: Refresh and rerun your Terraform configuration

Step 6.3.1

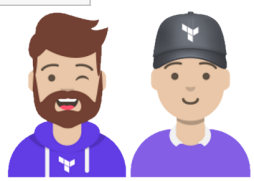
Rerun `terraform apply` to delete the original instance and recreate it once again. Now the public key will be installed on the new instance.

It may take a few minutes for the old instance to be destroyed and the new one crated. You might notice that both of these things happen in parallel:

```
...

aws_instance.web: Destroying... [id=i-00b20b227c41eca94]
module.server.aws_instance.web: Creating...
aws_instance.web: Still destroying... [id=i-00b20b227c41eca94, 10s elapsed]
module.server.aws_instance.web: Still creating... [10s elapsed]

...
```



Since there are no dependencies between the two, terraform can do both operations at the same time. This does mean that while the apply is still being run, both instances could exist at the same time, or neither might.

You'll also see that the outputs now include a list of (for now) one *public_dns* value and one *public_ip*:

```
...

Apply complete! Resources: 1 added, 0 changed, 1 destroyed.

Outputs:

public_dns = ec2-54-184-51-90.us-west-2.compute.amazonaws.com
public_ip = 54.184.51.90
```

When we moved the output configuration to the *server* module, we changed the definition of these outputs to be lists. This is so that we can update the module to create several instances at once in a future lab.