Terraform Module Sources

Modules can be sourced from a number of different locations, including both local and remote sources. The Terraform Module Registry, HTTP urls and S3 buckets are examples of remote sources, while folders and subfolders are examples of local sources. Support for various module sources allow you to include Terraform configuration from a variety of locations while still providing proper organization of code.

- Task 1: Source a local Terraform module
- Task 2: Explore the Public Module Registry and install a module
- Task 3: Source a module from GitHub

Task 1: Source a local Terraform module

While not required, local modules are commonly saved in a folder named `modules`, and each module is named for its respective function inside that folder. In support of this practice let's move our server module into a `modules` directory and observer the impact.

Step 1.1 - Refactor to use local modules directory

Create a `modules` directory and move your `server` directory into this directory as a subdirectory.

```
|-- modules
|   |-- server
|       |-- server.tf
```

Update the `source` of your module block to point to the modified source.

```
module "server" {
  source          = "./modules/server"
  ami             = data.aws_ami.ubuntu.id
  subnet_id       = aws_subnet.public_subnets["public_subnet_3"].id
  security_groups = [aws_security_group.vpc-ping.id, aws_security_group.ingress-ssh.id,
}

module "server_subnet_1" {
  source          = "./modules/server"
  ami             = data.aws_ami.ubuntu.id
  subnet_id       = aws_subnet.public_subnets["public_subnet_1"].id
  security_groups = [aws_security_group.vpc-ping.id, aws_security_group.ingress-ssh.id,
}
```


Any time that the source of a module is update, the working directory (root module) needs to be reintilized.

```
terraform init
```

We can follow that by issuing a `terraform plan` to make sure that moving our module source did not incur any changes to our infrastructure. This is expected because we have not changed any of the Terraform configuration, just the source of our module that was moved.

```
terraform plan

No changes. Your infrastructure matches the configuration.

Terraform has compared your real infrastructure against your configuration and found no
needed.
```

Step 1.2 - Create a new module source locally

Within our `modules` directory let's create another directory where we will include some enhancements to our `server` module. We will create these enhancments in a new directory called `web_server` and make a copy of our `server.tf` folder to this directory.

```
|-- modules
|   |-- server
|       |-- server.tf
|   |-- web_server
|       |-- server.tf
```

Let's update the `server.tf` inside our our `web_server` directory to now to allow us to provision our web application.

```
variable "ami" {}
variable "size" {
  default = "t2.micro"
}
variable "subnet_id" {}

variable "user" {}

variable "security_groups" {
  type = list(any)
}

variable "key_name" {
```


```
}

variable "private_key" {

}

resource "aws_instance" "web" {
  ami                         = var.ami
  instance_type               = var.size
  subnet_id                   = var.subnet_id
  vpc_security_group_ids      = var.security_groups
  associate_public_ip_address = true
  key_name                    = var.key_name
  connection {
    user        = var.user
    private_key = var.private_key
    host        = self.public_ip
  }


  provisioner "remote-exec" {
    inline = [
      "sudo rm -rf /tmp",
      "sudo git clone https://github.com/hashicorp/demo-terraform-101 /tmp",
      "sudo sh /tmp/assets/setup-web.sh",
    ]
  }

  tags = {
    "Name"        = "Web Server from Module"
    "Environment" = "Training"
  }

}

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

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


Step 1.2 - Update our `server_subnet_1` module to use the new `web_server` source

Let's now update our `server_subnet_1` module to use the new web_server source configuaration by updating the `source` arguement.

```
module "server_subnet_1" {
  source          = "./modules/web_server"
  ami             = data.aws_ami.ubuntu.id
  key_name        = aws_key_pair.generated.key_name
  user            = "ubuntu"
  private_key     = tls_private_key.generated.private_key_pem
  subnet_id       = aws_subnet.public_subnets["public_subnet_1"].id
  security_groups = [aws_security_group.vpc-ping.id, aws_security_group.ingress-ssh.id,
}
```

Any time that the source of a module is updated or added to our configuration, the working directory needs to be reintilized.

```
terraform init
```

You can make sure that the syntax is validate and formated properly by running the following:

```
terraform validate
terraform fmt -recursive
```

Step 1.2 - Apply our configuration with the updated module

Now that we have updated the source of our `server_subnet_1` module, we are ready to use it. Let's first see the ramifications of this change via a `terraform plan`

```
terraform plan
```

You will see that because we are now updating our server configuration via the new module to specify a key pair, that the server will need to be rebuilt.

```
Plan: 1 to add, 0 to change, 1 to destroy.
```

```
terraform apply
```

Now we will see that our new `web_server` module deploys a web application when used, versus the `server` module which simply deploys a generic server. We can of course switch between which modules are used by updating the `source` argument of our module - just be sure to run a `terraform plan` after making any changes to fully understand the impact of that change.

Task 2: Explore the Public Module Registry and install a module

Terraform Public Registry is an index of modules shared publicly. This public registry is the easiest way to get started with Terraform and find modules created by others in the community. You can also use a private registry as a feature of Terraform Cloud/Terraform Enterprise.

Modules on the public Terraform Registry can be sourced using a registry source address of the form //, with each module's information page on the registry site including the exact address to use.

We will use the AWS Autoscaling module to deploy an AWS Autoscaling group to our environment. Update your `main.tf` to include this module from the Terraform Module Registry.

```
module "autoscaling" {
  source  = "terraform-aws-modules/autoscaling/aws"
  version = "4.9.0"

  # Autoscaling group
  name = "myasg"

  vpc_zone_identifier = [aws_subnet.private_subnets["private_subnet_1"].id, aws_subnet.p
  min_size            = 0
  max_size            = 1
  desired_capacity    = 1

  # Launch template
  use_lt    = true
  create_lt = true

  image_id      = data.aws_ami.ubuntu.id
  instance_type = "t3.micro"

  tags_as_map = {
    Name = "Web EC2 Server 2"
  }

}
```

Any time that the source of a module is updated or added to our configuration, the working directory needs to be reintilized.

```
terraform init
```

We can follow that by issuing a `terraform plan` to see that additional resources that will be added by using the module.

```
terraform plan
```


Once we are happy with how the module is behaving we can issue a `terraform apply`

```
terraform apply
```

Task 3: Source a module from GitHub

Another source of modules that we can use are those that are published directly on GitHub. Terraform will recognize unprefixed github.com URLs and interpret them automatically as Git repository sources. Let's update the `source` of our autoscaling group module from the Public Module registry to use github.com instead. This will require us to remove the `version` argument from our module block.

```
module "autoscaling" {
  source = "github.com/terraform-aws-modules/terraform-aws-autoscaling"

  # Autoscaling group
  name = "myasg"

  vpc_zone_identifier = [aws_subnet.private_subnets["private_subnet_1"].id, aws_subnet.p
  min_size            = 0
  max_size            = 1
  desired_capacity    = 1

  # Launch template
  use_lt    = true
  create_lt = true

  image_id      = data.aws_ami.ubuntu.id
  instance_type = "t3.micro"

  tags_as_map = {
    Name = "Web EC2 Server 2"
  }

}
```

These GitHub schemes are treated as convenient aliases for the general Git repository address scheme, and so they obtain credentials in the same way and support the ref argument for selecting a specific revision. You will need to configure credentials in particular to access private repositories.

Now that our module source has been change we must reinitialize our working directory, plan and apply.

```
terraform init
```


We can follow that by issuing a `terraform plan` to see that additional resources that will be added by using the module.

```
terraform plan
```

Once we are happy with how the module is behaving we can issue a `terraform apply`

```
terraform apply
```