**Modules**

**Overview**

Modules are containers for multiple resources that are used together. A module consists of a collection of .tf and/or .tf.json files kept together in a directory.

Modules are the main way to package and reuse resource configurations with Terraform.

**The Root Module**

Every Terraform configuration has at least one module, known as its root module, which consists of the resources defined in the .tf files in the main working directory.

**Child Modules**

A Terraform module (usually the root module of a configuration) can call other modules to include their resources into the configuration. A module that has been called by another module is often referred to as a child module.

Child modules can be called multiple times within the same configuration, and multiple configurations can use the same child module.

**Published Modules**

In addition to modules from the local filesystem, Terraform can load modules from a public or private registry. This makes it possible to publish modules for others to use and to use modules that others have published.

The **Terraform Registry** hosts a broad collection of publicly available Terraform modules for configuring many kinds of common infrastructure. These modules are free to use, and Terraform can download them automatically if you specify the appropriate source and version in a module call block.

**What are modules for?**

Here are some of the ways that modules help solve the problems listed above:

**Organize configuration** - Modules make it easier to navigate, understand, and update your configuration by keeping related parts of your configuration together. Even moderately complex infrastructure can require hundreds or thousands of lines of configuration to implement. By using modules, you can organize your configuration into logical components.

**Encapsulate configuration**- Another benefit of using modules is to encapsulate configuration into distinct logical components. Encapsulation can help prevent unintended consequences, such as a change to one part of your configuration accidentally causing changes to other infrastructure, and reduce the chances of simple errors like using the same name for two different resources.

**Re-use configuration** - Writing all of your configuration from scratch can be time-consuming and error-prone. Using modules can save time and reduce costly errors by re-using configuration written either by yourself, other members of your team, or other Terraform practitioners who have published modules for you to use. You can also share modules that you have written with your team or the general public, giving them the benefit of your hard work.

**Provide consistency and ensure best practices** - Modules also help to provide consistency in your configurations. Not only does consistency make complex configurations easier to understand, it also helps to ensure that best practices are applied across all of your configurations. For instance, cloud providers give many options for configuring object storage services, such as Amazon S3 or Google Cloud Storage buckets. There have been many high-profile security incidents involving incorrectly secured object storage, and given the number of complex configuration options involved, it's easy to accidentally misconfigure these services.

Using modules can help reduce these errors. For example, you might create a module to describe how all of your organization's public website buckets will be configured, and another module for private buckets used for logging applications. Also, if a configuration for a type of resource needs to be updated, using modules allows you to make that update in a single place and have it be applied to all cases where you use that module.

**What is a Terraform module?**

A Terraform module is a set of Terraform configuration files in a single directory. Even a simple configuration consisting of a single directory with one or more .tf files is a module. When you run Terraform commands directly from such a directory, it is considered the **root module**. So in this sense, every Terraform configuration is part of a module. You may have a simple set of Terraform configuration files such as:

.

├── LICENSE

├── README.md

├── main.tf

├── variables.tf

├── outputs.tf

In this case, when you run terraform commands from within the minimal-module directory, the contents of that directory are considered the root module.

**Build a Module**

Terraform treats every configuration as a module. When you run terraform commands, the target directory containing Terraform configuration is treated as the root module.

We will create a module to create an IAM User for different departments using Terraform module.

Create a directory.

$ mkdir modules

Change into that directory in your terminal. Next, create a terraform config file name main.tf

$ cd modules

$ vim main.tf

#main.tf

provider "aws" {

region = "us-east-1"

}

resource "aws\_iam\_user" "my-new-user" {

name = "oliver-terraform-${var.environment}"

}

Create a variable file.

$ vim variables.tf

# variables.tf

variable "environment" {

default = "default"

}

Create an output file.

$ vim outputs.tf

# outputs.tf

output "my-terraform-user" {

value = aws\_iam\_user.my-new-user.name

}

Next, go to the previous folder and create a config file for the developer's department.

$ vim main.tf

Now create a config file and create an IAM user for the developers

# main.tf

module "usermodule" {

source = "./modules"

environment = "DEV"

}

Ensure that Terraform has downloaded all the necessary providers and modules by initializing it.

$ terraform init

Initializing modules...

- usermodule in modules

Initializing the backend...

Initializing provider plugins...

- Finding latest version of hashicorp/aws...

- Installing hashicorp/aws v3.27.0...

- Installed hashicorp/aws v3.27.0 (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!

You may now begin working with Terraform. Try running "terraform plan" to see

any changes that are required for your infrastructure. All Terraform commands

should now work.

If you ever set or change modules or backend configuration for Terraform,

rerun this command to reinitialize your working directory. If you forget,

    other

commands will detect it and remind you to do so if necessary.

Now that your new module is installed and configured, run terraform apply to provision your IAM User.

$ terraform apply

An execution plan has been generated and is shown below.

Resource actions are indicated with the following symbols:

+ create

Terraform will perform the following actions:

# module.usermodule.aws\_iam\_user.my-new-user will be created

+ resource "aws\_iam\_user" "my-new-user" {

+ arn = (known after apply)

+ force\_destroy = false

+ id = (known after apply)

+ name = "oliver-terraform-DEV"

+ path = "/"

+ unique\_id = (known after apply)

}

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

Do you want to perform these actions?

Terraform will perform the actions described above.

Only 'yes' will be accepted to approve.

Enter a value: yes

module.usermodule.aws\_iam\_user.my-new-user: Creating...

module.usermodule.aws\_iam\_user.my-new-user: Creation complete after 2s [id

    =oliver-terraform-DEV]

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

After running terraform apply, your new user will be created.

**⚠ Note:**

* After Terraform 0.12, only the root module outputs appear in state snapshots. Child module outputs now exist only temporarily in memory, due to them now being implemented consistently with input variables and local values. If you want to display output when you use a module, you can use an output block with the module block.
* output "example" {  
  value = module.usermodule.my-terraform-user  
  }

**⚠ Note:**

* Do not forget to destroy the resources you created with the command **terraform destroy**