**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 like this:

├── 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.

Calling modules

Terraform commands will only directly use the configuration files in one directory, which is usually the current working directory. However, your configuration can use module blocks to call modules in other directories. When Terraform encounters a module block, it loads and processes that module's configuration files.

A module that is called by another configuration is sometimes referred to as a "child module" of that configuration.

Local and remote modules

Modules can be loaded from either the local filesystem or a remote source. Terraform supports a variety of remote sources, including the Terraform Registry, most version control systems, HTTP URLs, and Terraform Cloud or Terraform Enterprise private module registries.

Module best practices

In many ways, Terraform modules are similar to the concepts of libraries, packages, or modules found in most programming languages, and they provide many of the same benefits. Just like almost any non-trivial computer program, real-world Terraform configurations should almost always use modules to provide the benefits mentioned above.

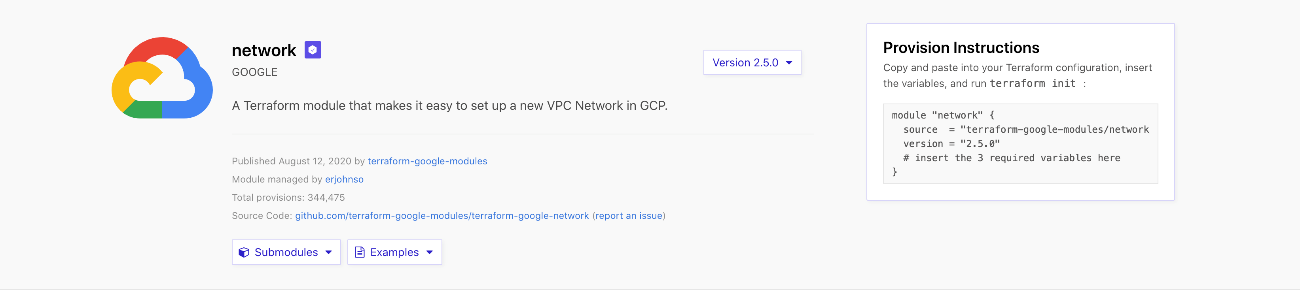
It is recommended that every Terraform practitioner use modules by following these best practices:

* Start writing your configuration with a plan for modules. Even for slightly complex Terraform configurations managed by a single person, the benefits of using modules outweigh the time it takes to use them properly.
* Use local modules to organize and encapsulate your code. Even if you aren't using or publishing remote modules, organizing your configuration in terms of modules from the beginning will significantly reduce the burden of maintaining and updating your configuration as your infrastructure grows in complexity.
* Use the public [Terraform Registry](https://registry.terraform.io/) to find useful modules. This way you can quickly and confidently implement your configuration by relying on the work of others.
* Publish and share modules with your team. Most infrastructure is managed by a team of people, and modules are an important tool that teams can use to create and maintain infrastructure. As mentioned earlier, you can publish modules either publicly or privately. You will see how to do this in a later lab in this series.

**Task 1. Use modules from the Registry**

In this section, you use modules from the [Terraform Registry](https://registry.terraform.io/) to provision an example environment in Google Cloud. The concepts you use here will apply to any modules from any source.

* Open the [Terraform Registry page](https://registry.terraform.io/modules/terraform-google-modules/network/google/3.3.0) for the Terraform Network module in a new browser tab or window. The page will look like this:



The page includes information about the module and a link to the source repository. The right side of the page includes a dropdown interface to select the module version and instructions for using the module to provision infrastructure.

When you call a module, the source argument is required. In this example, Terraform will search for a module in the Terraform Registry that matches the given string. You could also use a URL or local file path for the source of your modules. See the [Terraform documentation](https://www.terraform.io/docs/modules/sources.html) for a list of possible module sources.

The other argument shown here is the version. For supported sources, the version will let you define what version or versions of the module will be loaded. In this lab, you will specify an exact version number for the modules you use. You can read about more ways to specify versions in the [module documentation](https://www.terraform.io/docs/configuration/modules.html#module-versions).

Other arguments to module blocks are treated as input variables to the modules.

Create a Terraform configuration

1. To start, run the following commands in Cloud Shell to clone the example simple project from the Google Terraform modules GitHub repository and switch to the v6.0.1 branch:

git clone https://github.com/terraform-google-modules/terraform-google-network

cd terraform-google-network

git checkout tags/v6.0.1 -b v6.0.1

Copied!

content\_copy

This ensures that you're using the correct version number.

1. On the Cloud Shell toolbar, click **Open Editor**. To switch between Cloud Shell and the code editor, click **Open Editor** or **Open Terminal** as required, or click **Open in a new window** to leave the Editor open in a separate tab.
2. In the editor, navigate to terraform-google-network/examples/simple\_project, and open the main.tf file. Your main.tf configuration will look like this:

module "test-vpc-module" {

source = "terraform-google-modules/network/google"

version = "~> 6.0"

project\_id = var.project\_id # Replace this with your project ID

network\_name = "my-custom-mode-network"

mtu = 1460

subnets = [

{

subnet\_name = "subnet-01"

subnet\_ip = "10.10.10.0/24"

subnet\_region = "us-west1"

},

{

subnet\_name = "subnet-02"

subnet\_ip = "10.10.20.0/24"

subnet\_region = "us-west1"

subnet\_private\_access = "true"

subnet\_flow\_logs = "true"

},

{

subnet\_name = "subnet-03"

subnet\_ip = "10.10.30.0/24"

subnet\_region = "us-west1"

subnet\_flow\_logs = "true"

subnet\_flow\_logs\_interval = "INTERVAL\_10\_MIN"

subnet\_flow\_logs\_sampling = 0.7

subnet\_flow\_logs\_metadata = "INCLUDE\_ALL\_METADATA"

subnet\_flow\_logs\_filter = "false"

}

]

}

This configuration includes one important block:

* module "test-vpc-module" defines a Virtual Private Cloud (VPC), which will provide networking services for the rest of your infrastructure.

Set values for module input variables

Some input variables are *required*, which means that the module doesn't provide a default value; an explicit value must be provided in order for Terraform to run correctly.

* Within the module "test-vpc-module" block, review the input variables you are setting. Each of these input variables is documented in the [Terraform Registry](https://registry.terraform.io/modules/terraform-google-modules/network/google/3.3.0?tab=inputs). The required inputs for this module are:
  + network\_name: The name of the network being created
  + project\_id: The ID of the project where this VPC will be created
  + subnets: The list of subnets being created

In order to use most modules, you will need to pass input variables to the module configuration. The configuration that calls a module is responsible for setting its input values, which are passed as arguments to the module block. Aside from source and version, most of the arguments to a module block will set variable values.

On the Terraform Registry page for the Google Cloud network module, an Inputs tab describes all of the [input variables](https://registry.terraform.io/modules/terraform-google-modules/network/google/3.3.0?tab=inputs) that module supports.

Define root input variables

Using input variables with modules is very similar to how you use variables in any Terraform configuration. A common pattern is to identify which module input variables you might want to change in the future, and then create matching variables in your configuration's variables.tf file with sensible default values. Those variables can then be passed to the module block as arguments.

1. To retrieve your Project ID, run the following command in Cloud Shell:

gcloud config list --format 'value(core.project)'

Copied!

content\_copy

1. In the Editor, still in the same directory, navigate to variables.tf.
2. Fill in the variable project\_id with the output of the previous command. You must follow the format below and set the default value for the variable:

variable "project\_id" {

description = "The project ID to host the network in"

default = "FILL IN YOUR PROJECT ID HERE"

}

Copied!

content\_copy

1. In variables.tf, add in the variable network\_name. You can use the name example-vpc or any other name you'd like. You must follow the format below and set the default value for the variable:

variable "network\_name" {

description = "The name of the VPC network being created"

default = "example-vpc"

}

Copied!

content\_copy

1. Back in the main.tf file, update the network\_name parameter to use the variable you just defined by setting the value to var.network\_name.

module "test-vpc-module" {

...

project\_id = var.project\_id

network\_name = var.network\_name

...

Copied!

content\_copy

Define root output values

Modules also have output values, which are defined within the module with the output keyword. You can access them by referring to module.<MODULE NAME>.<OUTPUT NAME>. Like input variables, module outputs are listed under the outputs tab in the [Terraform Registry](https://registry.terraform.io/modules/terraform-google-modules/network/google/3.3.0?tab=inputs).

Module outputs are usually either passed to other parts of your configuration or defined as outputs in your root module. You will see both uses in this lab.

* Navigate to the outputs.tf file inside of your configuration's directory. Verify that the file contains the following:

output "network\_name" {

value = module.test-vpc-module.network\_name

description = "The name of the VPC being created"

}

output "network\_self\_link" {

value = module.test-vpc-module.network\_self\_link

description = "The URI of the VPC being created"

}

output "project\_id" {

value = module.test-vpc-module.project\_id

description = "VPC project id"

}

output "subnets\_names" {

value = module.test-vpc-module.subnets\_names

description = "The names of the subnets being created"

}

output "subnets\_ips" {

value = module.test-vpc-module.subnets\_ips

description = "The IP and cidrs of the subnets being created"

}

output "subnets\_regions" {

value = module.test-vpc-module.subnets\_regions

description = "The region where subnets will be created"

}

output "subnets\_private\_access" {

value = module.test-vpc-module.subnets\_private\_access

description = "Whether the subnets will have access to Google API's without a public IP"

}

output "subnets\_flow\_logs" {

value = module.test-vpc-module.subnets\_flow\_logs

description = "Whether the subnets will have VPC flow logs enabled"

}

output "subnets\_secondary\_ranges" {

value = module.test-vpc-module.subnets\_secondary\_ranges

description = "The secondary ranges associated with these subnets"

}

output "route\_names" {

value = module.test-vpc-module.route\_names

description = "The routes associated with this VPC"

}

Provision infrastructure

1. In Cloud Shell, navigate to your simple\_project directory:

cd ~/terraform-google-network/examples/simple\_project

Copied!

content\_copy

1. Initialize your Terraform configuration:

terraform init

Copied!

content\_copy

1. Create your VPC:

terraform apply

Copied!

content\_copy

1. To apply the changes and continue, respond to the prompt with **yes**.

Great! You've just used your first module. Your configuration's output should look like this:

Outputs:

network\_name = "example-vpc"

network\_self\_link = "https://www.googleapis.com/compute/v1/projects/qwiklabs-gcp-01-a68489b0625b/global/networks/example-vpc"

project\_id = ""

route\_names = []

subnets\_flow\_logs = [

false,

true,

true,

]

subnets\_ips = [

"10.10.10.0/24",

"10.10.20.0/24",

"10.10.30.0/24",

]

subnets\_names = [

"subnet-01",

"subnet-02",

"subnet-03",

]

....

....

Understand how modules work

When using a new module for the first time, you must run either terraform init or terraform get to install the module. When either of these commands is run, Terraform will install any new modules in the .terraform/modules directory within your configuration's working directory. For local modules, Terraform will create a symlink to the module's directory. Because of this, any changes to local modules will be effective immediately, without your having to re-run terraform get.

Clean up your infrastructure

Now you have seen how to use modules from the Terraform Registry, how to configure those modules with input variables, and how to get output values from those modules.

1. Destroy the infrastructure you created:

terraform destroy

Copied!

content\_copy

1. Respond to the prompt with yes. Terraform will destroy the infrastructure you created.
2. Once you've destroyed your resourced, delete the terraform-google-network folder.

rm -rd terraform-google-network -f

Copied!

content\_copy

Click *Check my progress* to verify the objective.

Provision infrastructure.

Check my progress

**Task 2. Build a module**

In the last task, you used a module from the Terraform Registry to create a VPC network in Google Cloud. Although using existing Terraform modules correctly is an important skill, every Terraform practitioner will also benefit from learning how to create modules. We recommend that you create every Terraform configuration with the assumption that it may be used as a module, because this will help you design your configurations to be flexible, reusable, and composable.

As you may already know, Terraform treats every configuration as a module. When you run terraform commands, or use Terraform Cloud or Terraform Enterprise to remotely run Terraform, the target directory containing Terraform configuration is treated as the root module.

In this task, you create a module to manage Compute Storage buckets used to host static websites.

Module structure

Terraform treats any local directory referenced in the source argument of a module block as a module. A typical file structure for a new module is:

├── LICENSE

├── README.md

├── main.tf

├── variables.tf

├── outputs.tf

**Note:**None of these files are required or has any special meaning to Terraform when it uses your module. You can create a module with a single .tf file or use any other file structure you like.

Each of these files serves a purpose:

* LICENSE contains the license under which your module will be distributed. When you share your module, the LICENSE file will let people using it know the terms under which it has been made available. Terraform itself does not use this file.
* README.md contains documentation in markdown format that describes how to use your module. Terraform does not use this file, but services like the Terraform Registry and GitHub will display the contents of this file to visitors to your module's Terraform Registry or GitHub page.
* main.tf contains the main set of configurations for your module. You can also create other configuration files and organize them in a way that makes sense for your project.
* variables.tf contains the variable definitions for your module. When your module is used by others, the variables will be configured as arguments in the module block. Because all Terraform values must be defined, any variables that don't have a default value will become required arguments. A variable with a default value can also be provided as a module argument, thus overriding the default value.
* outputs.tf contains the output definitions for your module. Module outputs are made available to the configuration using the module, so they are often used to pass information about the parts of your infrastructure defined by the module to other parts of your configuration.

Be aware of these files and ensure that you don't distribute them as part of your module:

* terraform.tfstate and terraform.tfstate.backup files contain your Terraform state and are how Terraform keeps track of the relationship between your configuration and the infrastructure provisioned by it.
* The .terraform directory contains the modules and plugins used to provision your infrastructure. These files are specific to an individual instance of Terraform when provisioning infrastructure, not the configuration of the infrastructure defined in .tf files.
* \*.tfvarsfiles don't need to be distributed with your module unless you are also using it as a standalone Terraform configuration because module input variables are set via arguments to the module block in your configuration.

**Note:**If you are tracking changes to your module in a version control system such as Git, you will want to configure your version control system to ignore these files. For an example, see this [.gitignore file](https://github.com/github/gitignore/blob/master/Terraform.gitignore) from GitHub.

Create a module

Navigate to your home directory and create your root module by constructing a new main.tf configuration file. Then create a directory called **modules** that contains another folder called gcs-static-website-bucket. You will work with three Terraform configuration files inside the gcs-static-website-bucket directory: website.tf, variables.tf, and outputs.tf.

1. Create the directory for your new module:

cd ~

touch main.tf

mkdir -p modules/gcs-static-website-bucket

Copied!

content\_copy

1. Navigate to the module directory and run the following commands to create three empty files:

cd modules/gcs-static-website-bucket

touch website.tf variables.tf outputs.tf

Copied!

content\_copy

1. Inside the gcs-static-website-bucket directory, run the following command to create a file called README.md with the following content:

tee -a README.md <<EOF

# GCS static website bucket

This module provisions Cloud Storage buckets configured for static website hosting.

EOF

Copied!

content\_copy

**Note:**Choosing the correct license for your modules is out of the scope of this lab. This lab will use the Apache 2.0 open source license.

1. Create another file called LICENSE with the following content:

tee -a LICENSE <<EOF

Licensed under the Apache License, Version 2.0 (the "License");

you may not use this file except in compliance with the License.

You may obtain a copy of the License at

http://www.apache.org/licenses/LICENSE-2.0

Unless required by applicable law or agreed to in writing, software

distributed under the License is distributed on an "AS IS" BASIS,

WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.

See the License for the specific language governing permissions and

limitations under the License.

EOF

Copied!

content\_copy

**Note:**Neither of these files is required or used by Terraform. Having them is a best practice for modules that might be shared with others.

Your current module directory structure should now look like this:

main.tf

modules/

└── gcs-static-website-bucket

├── LICENSE

├── README.md

├── website.tf

├── outputs.tf

└── variables.tf

1. Add this Cloud Storage bucket resource to your website.tf file inside the modules/gcs-static-website-bucket directory:

resource "google\_storage\_bucket" "bucket" {

name = var.name

project = var.project\_id

location = var.location

storage\_class = var.storage\_class

labels = var.labels

force\_destroy = var.force\_destroy

uniform\_bucket\_level\_access = true

versioning {

enabled = var.versioning

}

dynamic "retention\_policy" {

for\_each = var.retention\_policy == null ? [] : [var.retention\_policy]

content {

is\_locked = var.retention\_policy.is\_locked

retention\_period = var.retention\_policy.retention\_period

}

}

dynamic "encryption" {

for\_each = var.encryption == null ? [] : [var.encryption]

content {

default\_kms\_key\_name = var.encryption.default\_kms\_key\_name

}

}

dynamic "lifecycle\_rule" {

for\_each = var.lifecycle\_rules

content {

action {

type = lifecycle\_rule.value.action.type

storage\_class = lookup(lifecycle\_rule.value.action, "storage\_class", null)

}

condition {

age = lookup(lifecycle\_rule.value.condition, "age", null)

created\_before = lookup(lifecycle\_rule.value.condition, "created\_before", null)

with\_state = lookup(lifecycle\_rule.value.condition, "with\_state", null)

matches\_storage\_class = lookup(lifecycle\_rule.value.condition, "matches\_storage\_class", null)

num\_newer\_versions = lookup(lifecycle\_rule.value.condition, "num\_newer\_versions", null)

}

}

}

}

Copied!

content\_copy

The provider documentation is [GitHub](https://github.com/terraform-google-modules/terraform-google-cloud-storage/tree/master/modules/simple_bucket).

1. Navigate to the variables.tf file in your module and add the following code:

variable "name" {

description = "The name of the bucket."

type = string

}

variable "project\_id" {

description = "The ID of the project to create the bucket in."

type = string

}

variable "location" {

description = "The location of the bucket."

type = string

}

variable "storage\_class" {

description = "The Storage Class of the new bucket."

type = string

default = null

}

variable "labels" {

description = "A set of key/value label pairs to assign to the bucket."

type = map(string)

default = null

}

variable "bucket\_policy\_only" {

description = "Enables Bucket Policy Only access to a bucket."

type = bool

default = true

}

variable "versioning" {

description = "While set to true, versioning is fully enabled for this bucket."

type = bool

default = true

}

variable "force\_destroy" {

description = "When deleting a bucket, this boolean option will delete all contained objects. If false, Terraform will fail to delete buckets which contain objects."

type = bool

default = true

}

variable "iam\_members" {

description = "The list of IAM members to grant permissions on the bucket."

type = list(object({

role = string

member = string

}))

default = []

}

variable "retention\_policy" {

description = "Configuration of the bucket's data retention policy for how long objects in the bucket should be retained."

type = object({

is\_locked = bool

retention\_period = number

})

default = null

}

variable "encryption" {

description = "A Cloud KMS key that will be used to encrypt objects inserted into this bucket"

type = object({

default\_kms\_key\_name = string

})

default = null

}

variable "lifecycle\_rules" {

description = "The bucket's Lifecycle Rules configuration."

type = list(object({

# Object with keys:

# - type - The type of the action of this Lifecycle Rule. Supported values: Delete and SetStorageClass.

# - storage\_class - (Required if action type is SetStorageClass) The target Storage Class of objects affected by this Lifecycle Rule.

action = any

# Object with keys:

# - age - (Optional) Minimum age of an object in days to satisfy this condition.

# - created\_before - (Optional) Creation date of an object in RFC 3339 (e.g. 2017-06-13) to satisfy this condition.

# - with\_state - (Optional) Match to live and/or archived objects. Supported values include: "LIVE", "ARCHIVED", "ANY".

# - matches\_storage\_class - (Optional) Storage Class of objects to satisfy this condition. Supported values include: MULTI\_REGIONAL, REGIONAL, NEARLINE, COLDLINE, STANDARD, DURABLE\_REDUCED\_AVAILABILITY.

# - num\_newer\_versions - (Optional) Relevant only for versioned objects. The number of newer versions of an object to satisfy this condition.

condition = any

}))

default = []

}

Copied!

content\_copy

1. Add an output to your module in the outputs.tf file inside your module:

output "bucket" {

description = "The created storage bucket"

value = google\_storage\_bucket.bucket

}

Copied!

content\_copy

Like variables, outputs in modules perform the same function as they do in the root module but are accessed in a different way. A module's outputs can be accessed as read-only attributes on the module object, which is available within the configuration that calls the module.

1. Return to the main.tf in your **root directory** and add a reference to the new module:

module "gcs-static-website-bucket" {

source = "./modules/gcs-static-website-bucket"

name = var.name

project\_id = var.project\_id

location = "us-east1"

lifecycle\_rules = [{

action = {

type = "Delete"

}

condition = {

age = 365

with\_state = "ANY"

}

}]

}

Copied!

content\_copy

1. In your **root directory**, create an outputs.tf file for your root module:

cd ~

touch outputs.tf

Copied!

content\_copy

1. Add the following code in the outputs.tf file:

output "bucket-name" {

description = "Bucket names."

value = "module.gcs-static-website-bucket.bucket"

}

Copied!

content\_copy

1. In your **root directory**, create a variables.tf file:

touch variables.tf

Copied!

content\_copy

1. Add the following code to the variables.tf file. Set the variables project\_id and name to your

variable "project\_id" {

description = "The ID of the project in which to provision resources."

type = string

default = "FILL IN YOUR PROJECT ID HERE"

}

variable "name" {

description = "Name of the buckets to create."

type = string

default = "FILL IN A (UNIQUE) BUCKET NAME HERE"

}

Copied!

content\_copy

**Note:**The name of your storage bucket must be globally unique. Using your name and the date is usually a good way to create a unique bucket name. You can also use your Project ID.

Install the local module

Whenever you add a new module to a configuration, Terraform must install the module before it can be used. Both the terraform get and terraform init commands will install and update modules. The terraform init command will also initialize backends and install plugins.

1. Install the module:

terraform init

Copied!

content\_copy

1. Provision your bucket:

terraform apply

Copied!

content\_copy

1. Respond **yes** to the prompt. Your bucket and other resources will be provisioned.

Upload files to the bucket

You have now configured and used your own module to create a static website. You may want to visit this static website. Right now there is nothing inside your bucket, so there is nothing to see at the website. In order to see any content, you will need to upload objects to your bucket. You can upload the contents of the www directory in the GitHub repository.

1. Download the sample contents to your home directory:

cd ~

curl https://raw.githubusercontent.com/hashicorp/learn-terraform-modules/master/modules/aws-s3-static-website-bucket/www/index.html > index.html

curl https://raw.githubusercontent.com/hashicorp/learn-terraform-modules/blob/master/modules/aws-s3-static-website-bucket/www/error.html > error.html

Copied!

content\_copy

1. Copy the files over to the bucket, replacing YOUR-BUCKET-NAME with the name of your storage bucket:

gsutil cp \*.html gs://YOUR-BUCKET-NAME

Copied!

content\_copy

1. In a new tab in your browser, go to the website https://storage.cloud.google.com/YOUR-BUCKET-NAME/index.html, replacing YOUR-BUCKET-NAME with the name of your storage bucket.

You should see a basic HTML web page that says **Nothing to see here**.

Click *Check my progress* to verify the objective.

Build a module.

Check my progress

Clean up the website and infrastructure

Lastly, you will clean up your project by destroying the infrastructure you just created.

1. Destroy your Terraform resources:

terraform destroy

Copied!

content\_copy

After you respond to the prompt with yes, Terraform will destroy all of the resources you created by following this lab.

**Congratulations!**

In this lab, you learned the foundations of Terraform modules and how to use a pre-existing module from the Registry. You then built your own module to create a static website hosted on a Cloud Storage bucket. In doing so, you defined inputs, outputs, and variables for your configuration files and learned the best-practices for building modules.