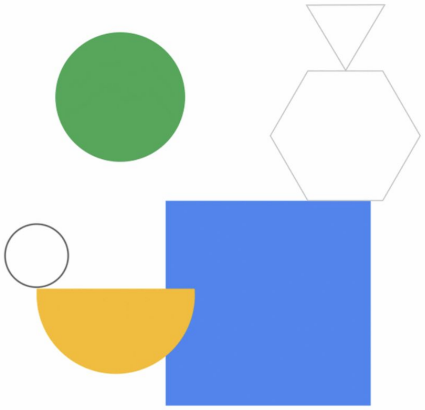
Google Cloud

Organizing and Reusing Configuration with Terraform Modules



Proprietary + Confidential

Objectives

After you complete this module, you will be able to:

01 Define Terraform modules.

02 Use modules to reuse configurations.

03 Use modules from the public registry.

04 Use input variables to parameterize configurations.

05 Use output values to access resource attributes outside the module.

Google Cloud

Welcome to the module called “Organizing and Reusing Configuration with Terraform Modules”. You will explore what modules are, how to use them from a public registry, how to use modules to reuse configurations, and parameterize configurations using input variables. You will also explore how to use output values to access resource attributes outside of the module.

Proprietary + Confidential

Topics

01	Need for modules
02	Modules overview
03	Example of a module, use cases and benefits
04	Reuse configurations by using modules
05	Use variables to parameterize a module
06	Pass resource attribute outside the module by using output values
07	Modules best practices
08	A real time scenario

Proprietary + Confidential

Problem: Updating repeated code

Web Server

- VM image
- Machine type
- Static IP
- Service account

```
resource "google_compute_instance" "serverVM" {
#All necessary parameters defined
   machine_type = "f1-micro"
}

resource “google_compute_address” “static_ip”{
..
}
resource "google_compute_disk" "server_disk" {
..
}

resource "google_service_account" "service_account" {
..
}
```

Google Cloud

Before we explore what modules are, let’s use a simple scenario to see why modules are necessary.

Consider a scenario where you have to create a web server in a custom network. For example, the typical attributes that represent the web server are:

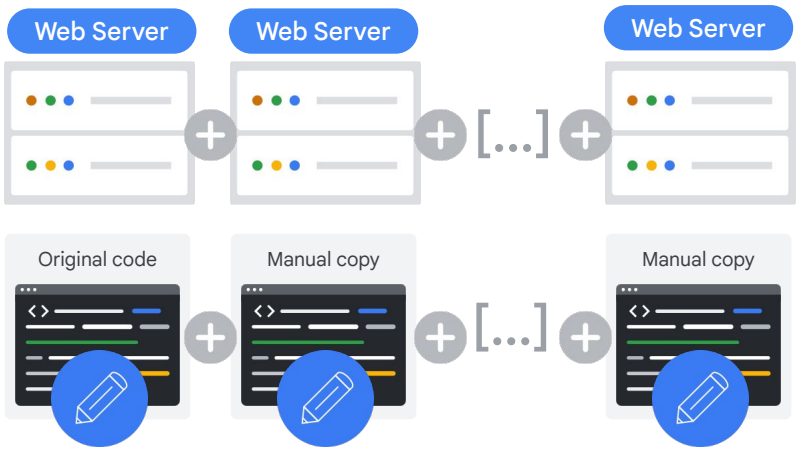
- The machine type
- Disk
- Static IP address
- Google service account

The associated code to deploy a server with these attributes is similar to the one shown on the slide.

Proprietary + Confidential

Disadvantages of code repetition

- Unmanageable
- Error prone
- Inefficient



Google Cloud

What if you have to deploy many servers of same kind? You might have to manually copy the code. What if you have to update one attribute for all these servers? Then you have to manually find that attribute in every occurrence and manually update.

The disadvantages of manually copying the code are:

- As new resources or changes are included, the code becomes bigger, unmanageable, and harder to read.
- Copying the code becomes a cumbersome process when every little change made to the code has to be applied with no errors across all the environments involved.
- It also makes the code inefficient as similar blocks are duplicated and can cause discrepancies when you are updating those parts of the configuration across the environments.

Proprietary + Confidential

Solution: Create modules

```
-- server/
   -- main.tf
   -- outputs.tf
   -- variables.tf
```

- DRY (don't repeat yourself): Replace repeated code with abstraction to avoid redundancy.
- Define the reusable code within a module named server, so that any change you make to the module is reflected across all the environments that you plan to reuse.

```
resource "google_compute_instance" "serverVM" {
#All necessary parameters defined
   machine_type = "f1-micro"
   boot_disk {
    initialize_params {
      image = "debian-cloud/debian-9"
    }
  }
}

resource “google_compute_address” “static_ip”{
..
}
resource "google_compute_network" "mynetwork" {
..
}

resource "google_compute_firewall" "default" {
..
}
```

Google Cloud

In the programming world, this problem is addressed with the principle of DRY, which means “Don’t repeat yourself.” The idea of DRY is that you don't need to repeat the same set of codes multiple times because you can replace it with abstraction to avoid redundancy. This practice encourages building efficient codes that are readable and reusable. In general-purpose programming languages like Ruby, Java, and Python, functions are used to implement the DRY principle. Any piece of code that has to be copied at multiple places is placed inside a function and reused across the main code. With Terraform, you can place your reusable code inside a Terraform module and reuse that module at multiple places across the code. We will cover how to create and call a module in the upcoming section.

Define the reusable code within a module named server, so that any change you make to the module is reflected across all the environments that you plan to reuse.

Proprietary + Confidential

Topics

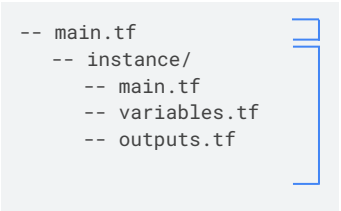
01	Need for modules
02	Modules overview
03	Example of a module, use cases and benefits
04	Reuse configurations by using modules
05	Use variables to parameterize a module
06	Pass resource attribute outside the module by using output values
07	Modules best practices
08	A real time scenario

Proprietary + Confidential

A module is collection of configuration files

- One or more Terraform configuration files (.tf) in a directory can form a module.
- Modules let you group a set of resources together and reuse them later.
- The root module consists of the .tf files that are stored in your working directory where you run `terraform plan` or `terraform apply`.

Note: The root module is where other modules and resources are instantiated.



Google Cloud

Any Terraform configuration file (.tf file) in a directory, even only one, forms a module. Modules allow you group a set of resources together and reuse them later. A module can be referenced from other module. So far, we have unintentionally written our terraform configuration in a module called the root module. The directory from which you run the terraform command is considered the root module. Notice that the root module consists of the .tf files that are stored in your working directory where you run `terraform plan` and `terraform apply`.

The root module is where other modules and resources are instantiated.

Proprietary + Confidential

Topics

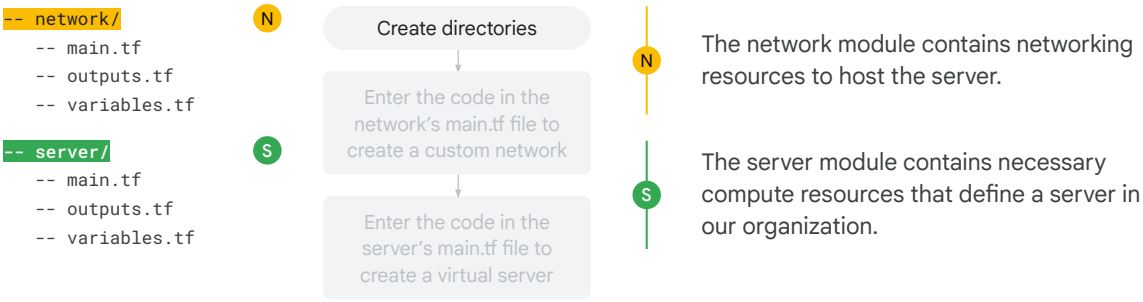
01	Need for modules
02	Modules overview
03	Example of a module, use cases and benefits
04	Reuse configurations by using modules
05	Use variables to parameterize a module
06	Pass resource attribute outside the module by using output values
07	Modules best practices
08	A real time scenario

Google Cloud

Let us take a look at how to create a module with an example.

Proprietary + Confidential

Write your first module



Google Cloud

As a starting point, we create an example that contains two modules called “server” and “network”, and we’ll build on this example as we advance. The network module will contain networking resources to host the server. The server module will contain necessary compute resources that define a server.

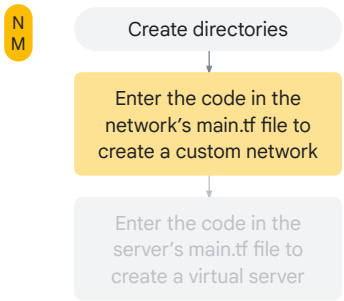
To create a module, first create a directory of Terraform files. In our example, we create two directories, named network and server. Each directory has its own main.tf file.

Proprietary + Confidential

Write your first module

```
-- network/
   -- main.tf
   -- outputs.tf
   -- variables.tf

-- server/
   -- main.tf
   -- outputs.tf
   -- variables.tf
```




```
resource "google_compute_network" "mynetwork" {
  name                    = "mynetwork"
  auto_create_subnetworks  = true
  routing_mode             = global
  mtu                     = 1460
}

resource "google_compute_firewall" "default" {
#All necessary parameters defined
}
```

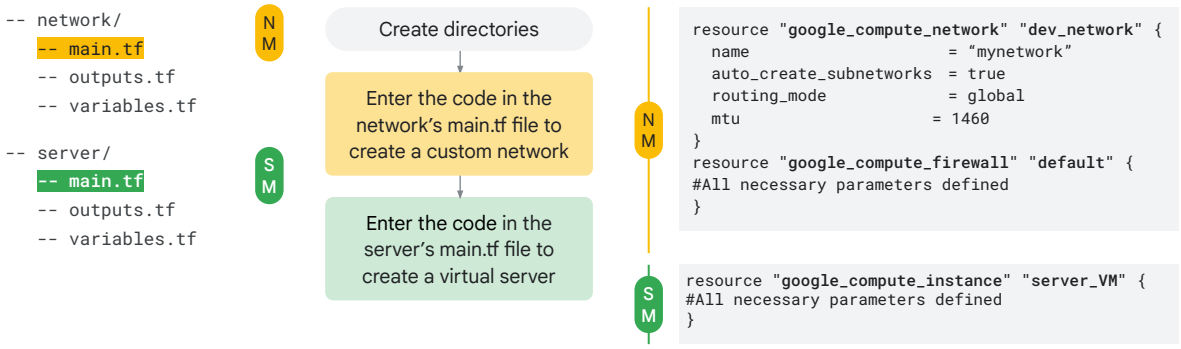
Notice that multiple resources are grouped in the network module.

Google Cloud

After you created the files, write the code to create a custom network in the main.tf file within the network directory. Notice that multiple resources are grouped in the network module.

Proprietary + Confidential

Write your first module

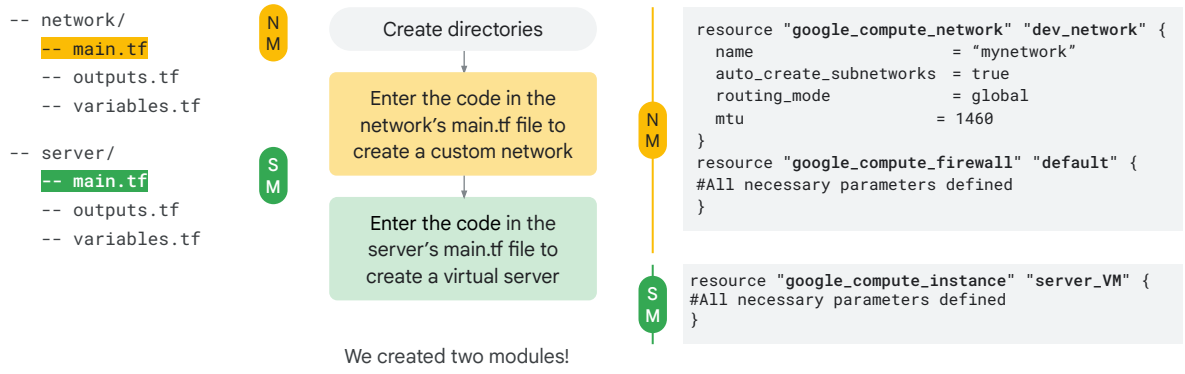


Google Cloud

In the main.tf file of the server directory, enter the code associated to create a virtual server. Enter all the code associated with the server within the main.tf file.

Proprietary + Confidential

Write your first module

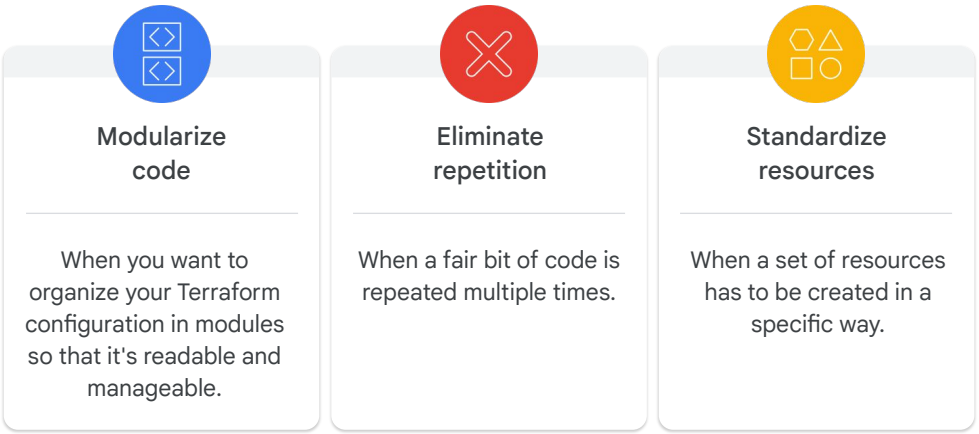


Google Cloud

We just created 2 modules. Note that at this stage, we have only created the modules and written the configuration code. This process does not create the infrastructure resources. These module are instantiated when it is called and you execute the `terraform apply` command.

Proprietary + Confidential

Modules Use Cases



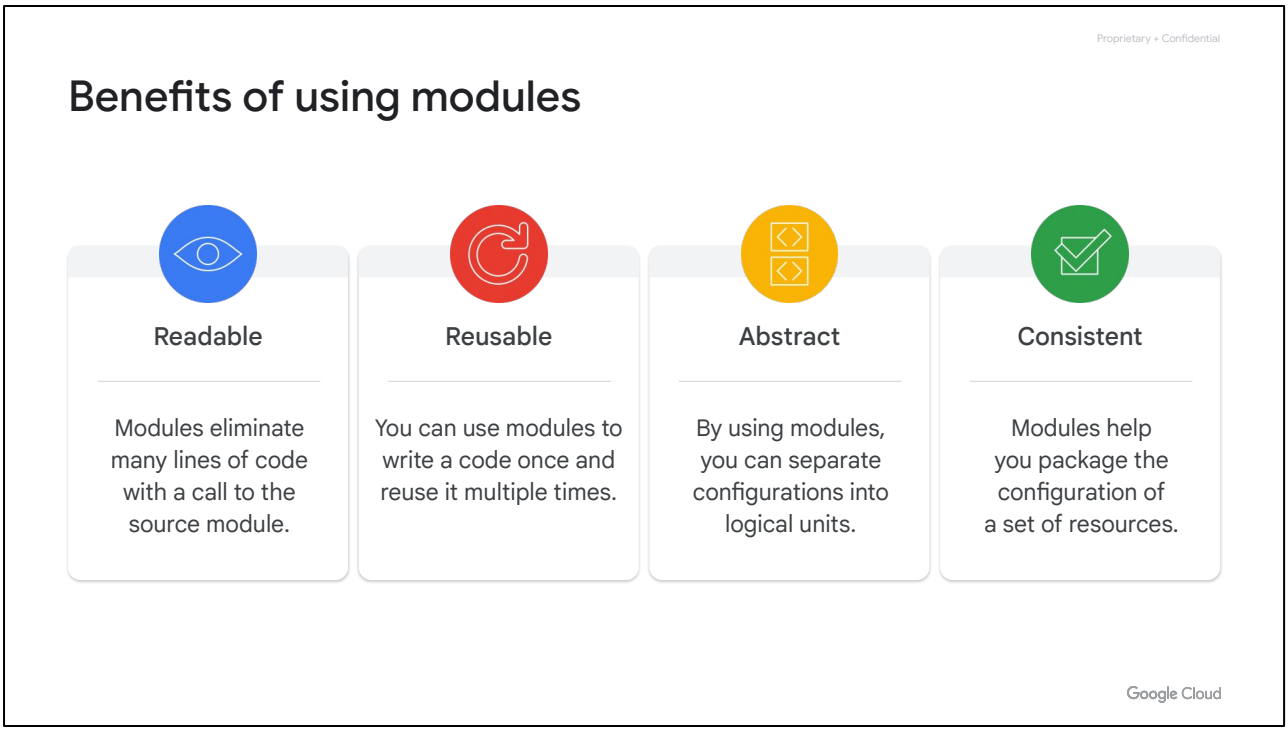
Google Cloud

Some of the reasons to use modules are to modularize the code, eliminate repetition and standardize resource creatio. Let us go over each of the use cases in detail.
Modularize code: By now you you're probably aware that Terraform is a tool to manage your infrastructure as code. But imagine having a single Terraform configuration file or a .tf file for managing your entire cloud environment. Sounds challenging? If you have a single Terraform file for all the infrastructure components within your environment, the code will be unreadable. As your business scales up, the number of lines in your Terraform code should also grows in parallel.

Thus you should break your code into reusable modules so that it is readable and easy to manage.

Eliminate repetition: When a set of resources is repeated, it makes the code longer and difficult to read. You modules to remove repetition of code.

Standardize resource creation: When a set of resources has to be configured in the same way every time it is created, you can standardize the configuration by placing the code in a module. In this way, when the module is called, the same source code will be used; standardizing the creation of resources. For example, if you want to deploy database servers with encrypted disks, you can ensure that the disks will always be encrypted by hardcoding the encryption configuration within the module and calling this module when a disk has to be created.



The modular approach to code management makes the configuration:

- Readable: Modules make the code easy to read. Modules eliminate many lines of code with a call to the source module.
- Reusable: You can use modules to write a code once and reuse it multiple times across various environments
- Abstract: By using modules, you can also separate configurations into logical units; reducing their dependency and making them easier to debug.
- Consistent: Modules help you package the configuration of a set of resources; enabling consistent replication of the resource.

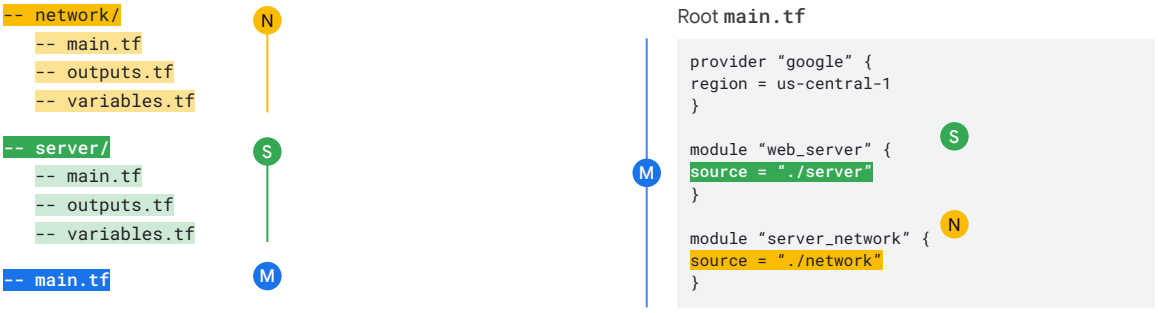
Proprietary + Confidential

Topics

01	Need for modules
02	Modules overview
03	Example of a module, use cases and benefits
04	Reuse configurations by using modules
05	Use variables to parameterize a module
06	Pass resource attribute outside the module by using output values
07	Modules best practices
08	A real time scenario

Proprietary + Confidential

Calling the module to reuse the configuration



You can use the module by calling it in your main configuration.

Google Cloud

Now that we have defined the modules. The next step is to call the module from the parent main.tf file. To call a module means to reuse it in your configuration. You can call the module to reference the code in the module block. In this example, the main.tf is the calling module. The parent main.tf file uses the source argument to call the server and network module. Run the `terraform init` command to download any modules referenced by a configuration.

Proprietary + Confidential

`source` meta argument

- `source` is a meta argument, whose value provides the path to the configuration code.
- The value can be a local or remote path.
- There are several supported remote source types, such as Terraform Registry, GitHub, Bitbucket, HTTP URLs, and Cloud Storage buckets.

Syntax for calling the module

```
module "<NAME>" {
 source = "<source_location>"
 [CONFIG ...]
}
```

Google Cloud

The `source` argument determines the location of the module source code.

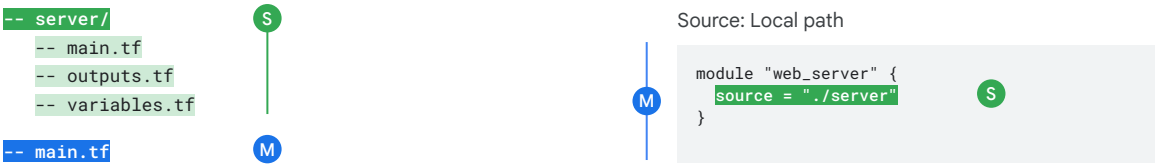
Every module you call within Terraform requires a mandatory `source` argument, which is a meta-argument within the module block. This value can either be a local path within the root directory or a remote path to a module source that Terraform downloads. The different source types supported by terraform are:

The different source types supported by Terraform are Terraform Registry, GitHub, Bitbucket, HTTP URLs, GCS buckets, etc.

In this course, we explore local paths, Terraform Registry, and GitHub.

Proprietary + Confidential

Module source: Local path



Google Cloud

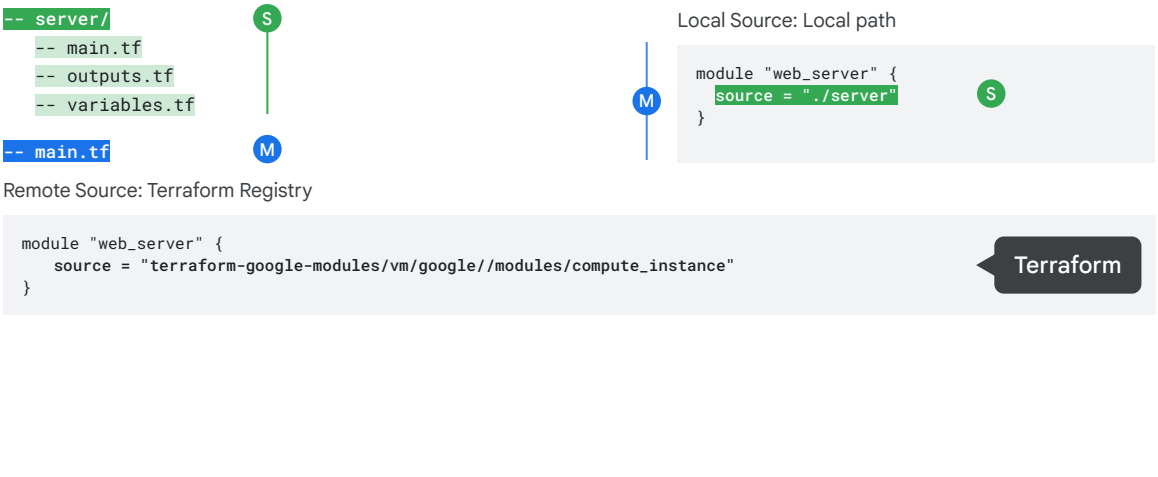
Let’s start with the local path.

Local path is used to reference a module stored within the same directory as the calling module.

For example, if the module that you want to call is stored in a directory named “servers” that is located in the same place as your root module directory, your root configuration will be as shown on the slide. A local path starts with either ./ or ../. Local paths are unique when compared to other module sources, because they do not require any installation. The files are locally referenced from the child module to the parent module directly. As a consequence, no explicit update is required.

Proprietary + Confidential

Module source: Terraform Registry



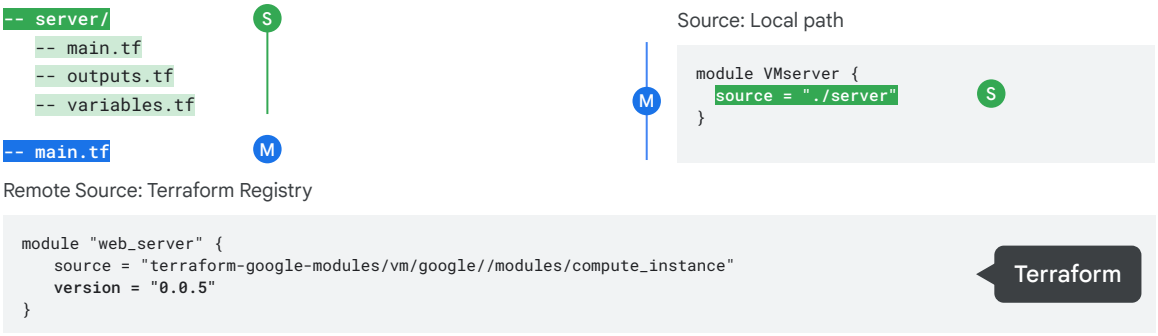
Google Cloud

Terraform Registry
Terraform Registry is one of the commonly used module sources. It contains a directories of publicly usable modules for various infrastructure components such as load balancers, SQL instances, etc. They are extremely useful for complex deployments.

The registry source address has to be in the format `<NAMESPACE>/<NAME>/<PROVIDER>` . To use the code published in a Terraform Registry for the Google Cloud provider, use the format `terraform-google-modules/gcloud/google` .

Proprietary + Confidential

Terraform Registry: Version Constraint



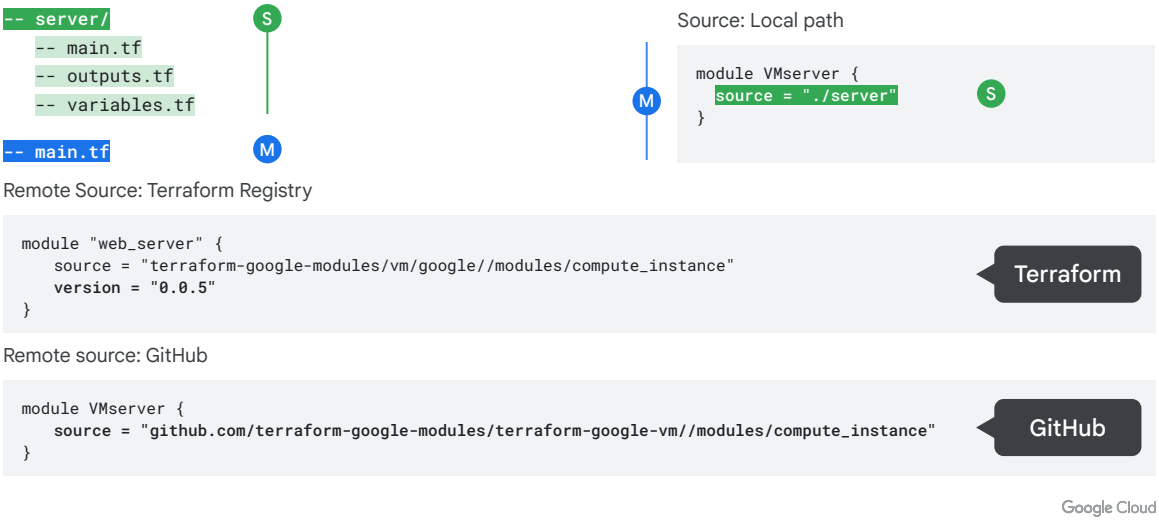
Google Cloud

Providing version constraint is also recommended to avoid any unwanted changes to the Terraform configuration that reuses the modules. The version argument can be assigned a version string. This allows Terraform to automatically upgrade the module to new patch releases while still keeping a solid target. Only the modules installed from the Terraform Registry support this constraint.

Refer to the Terraform documentation for more information on Terraform Registry.

Proprietary + Confidential

Module source: GitHub



Google Cloud

The next most commonly used remote source after Terraform Registry is the GitHub repository.

Github

Similar to Terraform Registry, you can directly enter the GitHub URL where the source code is located.

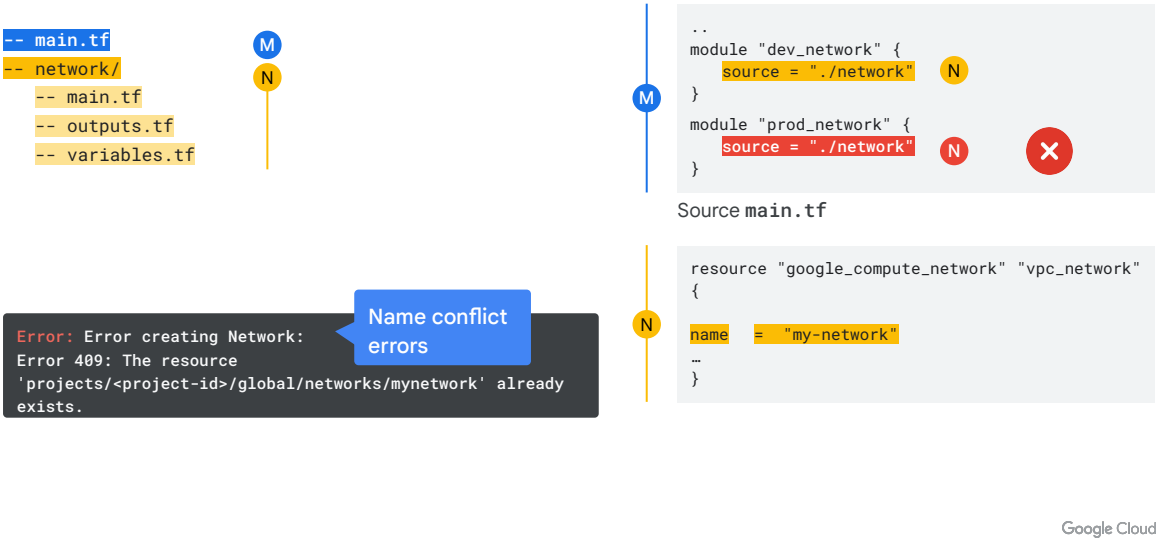
Proprietary + Confidential

Topics

01	Need for modules
02	Modules overview
03	Example of a module, use cases and benefits
04	Reuse configurations by using modules
05	Use variables to parameterize a module
06	Pass resource attribute outside the module by using output values
07	Modules best practices
08	A real time scenario

Proprietary + Confidential

Eliminate hard coding by using variables



Google Cloud

Notice that the examples that we covered so far had all the attributes of the resources defined within the module are hardcoded.. For example, the name of the network is hardcoded within the main.tf file of the server module. So if you use this module more than once, you will receive name conflict errors as shown on the screen. In such scenarios, you should have the flexibility to configure a few attributes differently in different environments and the capability to standardize a few of them.

Another example, where variables can be quite useful is when you are working with different environment such as development, production and staging. You should run a small machine type in the staging environment and a larger machine type in the production environment.

Proprietary + Confidential

Parameterize your configuration with input variables

```
-- network/
   -- main.tf
   -- outputs.tf
   -- variables.tf

-- main.tf
```

Replace the hard coded arguments with a variable.

Declare the variables in the variables.tf file

Pass the value for the input variable when you call the module.

```
resource "google_compute_network" "vpc_network"
{
  name    = var.network_name
..
}
```

Google Cloud

Now let us see how to parameterize the name argument. Start by replacing the hardcoded values within your module with a variable. This can be done by assigning the argument you wish to parameterize, in this case network name, by using the format var.<variable_name>. This provides the flexibility to configure the name argument when we call the module.

Proprietary + Confidential

Parameterize your configuration with input variables

```
-- network/
   -- main.tf
   -- outputs.tf
   -- variables.tf

-- main.tf
```

N V

Replace the hard coded arguments with a variable.

Declare the variables in the variables.tf file

Pass the value for the input variable when you call the module.

N V

```
variable "network_name" {
type          = string
description   = “name of the network”
}
```

Google Cloud

Then, use the variable block to declare the attributes that are parameterized within the module. Place the variable declared in a variables.tf file within the same directory where the resource is defined. Ensure to specify the appropriate variable type.

Proprietary + Confidential

Parameterize your configuration with input variables

```
-- network/
   -- main.tf
   -- outputs.tf
   -- variables.tf

-- main.tf
```

M

Replace the hard coded arguments with a variable.

↓

Declare the variables in the variables.tf file

↓

Pass the value for the input variable when you call the module.

M

```
..
module "dev_network" {
   source            = "./network"
   network_name      = "my-network1"
}

module "prod_network" {
   source            = "./network"
   network_name      = "my-network2"
}
```

You cannot pass values to variables for modules at run time.

Google Cloud

You can pass the value to the input variable when you call the module. As we have parameterized network name in our example, you can now reuse the same network module twice in the main configuration and provide different name for each instance. Remember to run the terraform init command to download any modules referenced in a configuration.

Note that unlike root configuration, you cannot pass values to variable for modules at run time.

Proprietary + Confidential

Topics

01	Need for modules
02	Modules overview
03	Example of a module, use cases and benefits
04	Reuse configurations by using modules
05	Use variables to parameterize a module
06	Pass resource attribute outside the module by using output values
07	Modules best practices
08	A real time scenario

Proprietary + Confidential

Pass resource attributes outside the module

```
-- main.tf
   -- network/
      -- main.tf
      -- outputs.tf
      -- variables.tf

   -- server/
      -- main.tf
      -- outputs.tf
      -- variables.tf
```

The server module needs the network name created by the network module.

N

```
resource "google_compute_network" "my_network"
{
  name                    = “mynetwork”
  auto_create_subnetworks = true
  routing_mode            = “GLOBAL”
  mtu                     = 1460
}
```

S

```
resource "google_compute_instance" "server_VM"
{
network = <network created by network module>
#All necessary parameters defined
}

resource "google_compute_firewall" "default" {
#All necessary parameters defined
}
```

Note: Use output values to pass resources attributes between modules

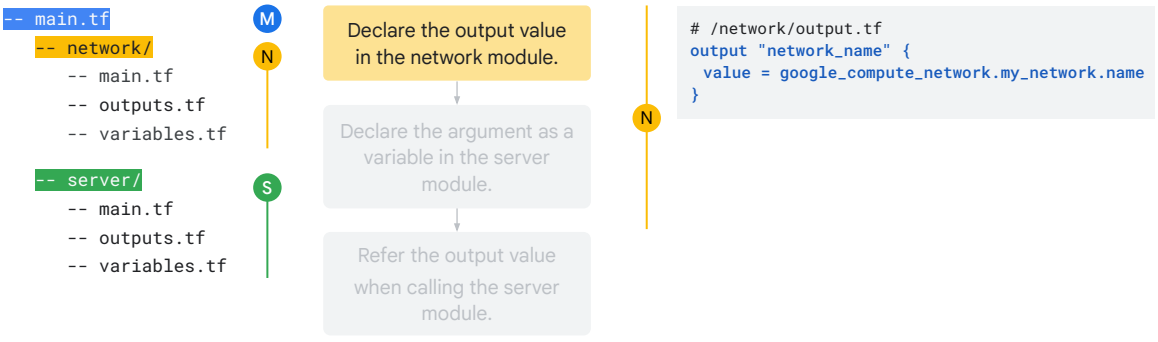
Google Cloud

If we want to pass arguments from a resource in one module to another, you will have to explicitly configure the argument as an output value in Terraform. For example, the network module includes the network name.

The server module should receive the network name from the module that defines the network. If we have configured this in two separate modules, then we can pass this value by using the output values.

Proprietary + Confidential

Using output values

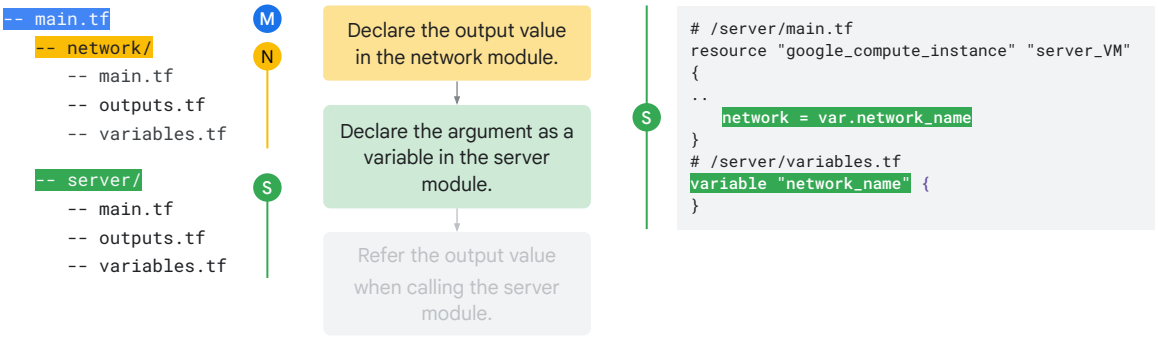


Google Cloud

To pass the network name form the network module, declare the network name as the output value. This exposes the resource attribute outside of the module.

Proprietary + Confidential

Using output values

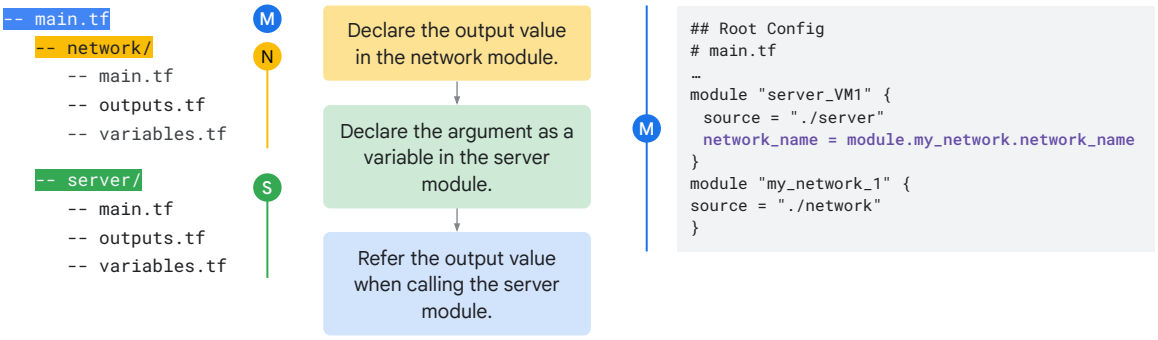


Google Cloud

Define the network name as a variable in the server module, so that it can accept the values passed outside the module.

Proprietary + Confidential

Using output values

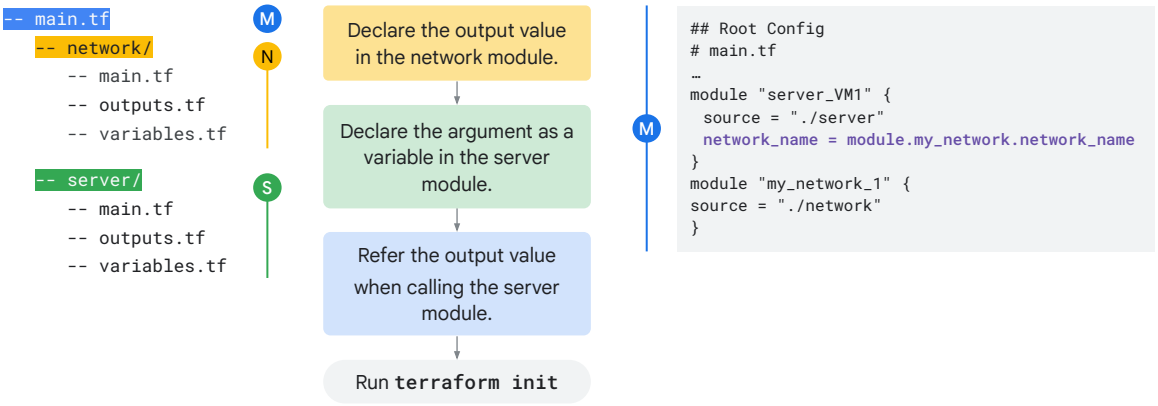


Google Cloud

Now in the main configuration, reference the output value when calling the server module. You can refer the output value by using the format `module.<module_name>.<output_value>` . In our example, we refer to the output value of the my_network module using `module.my_network.network_name` .
Notice that both modules are called from the root configuration. Each time a module is instantiated, `terraform init` has to be executed.

Proprietary + Confidential

Using output values



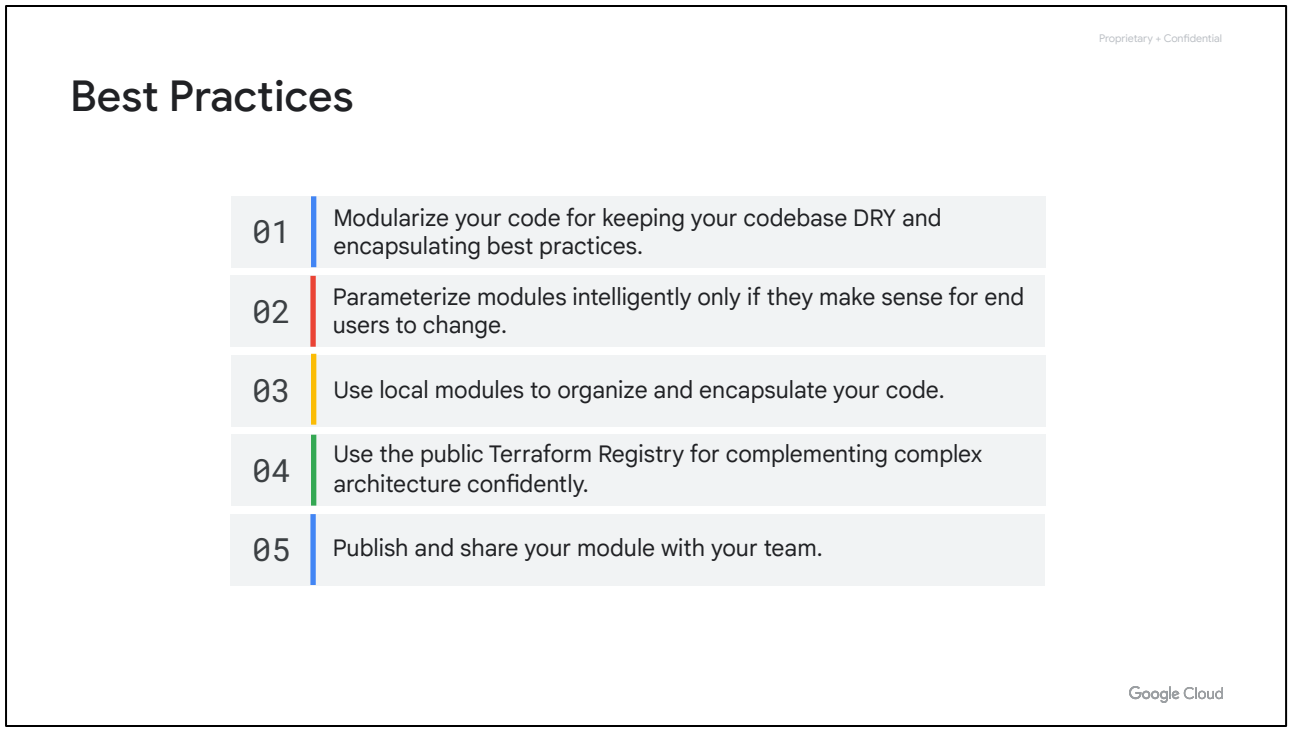
Google Cloud

Once you complete step 3 and run `terraform init`, you will be able to pass the network name from network module to the server module.

Proprietary + Confidential

Topics

01	Need for modules
02	Modules overview
03	Example of a module, use cases and benefits
04	Reuse configurations by using modules
05	Use variables to parameterize a module
06	Pass resource attribute outside the module by using output values
07	Modules best practices
08	A real time scenario



Every Terraform practitioner should use modules by following these best practices:

1. Don’t over-use modules. You should modularize your code for keeping your codebase DRY and encapsulating best practices. Don’t be obsessed with eliminating all the duplications in your configations: sometimes it’s fine to include a bit of duplication to allow for more explicit configuration. It helps someone reading the Terraform code to visualize the infrastructure. In cases where you want to loop over multiple values, use the built-in features of Terraform such as count or for_each instead of building your own custom scripting.
2. Parameterize modules intelligently only if they make sense for end users to change. For example, in a custom network module, the MTU and routing_mode can be standardized, but its name has to be parameterized to ensure reusability. When considering parameterization, focus on the values that you must change. If a value is always fixed in your environment, hardcoding is fine.
3. Use local modules to organize and encapsulate your code. Even if you don't use or publish remote modules, using modules to organize your configuration from the beginning will significantly reduce the burden of maintaining and updating your configuration as your infrastructure grows in complexity.
4. Use the public Terraform Registry to find useful modules. This way you can quickly and confidently implement your configuration by relying on the work of others.
5. Publish and share modules with your team. Most infrastructure is managed by a team of people, and modules are an important tool that they can use to create and maintain infrastructure. As mentioned earlier, you can publish modules either publicly or privately.

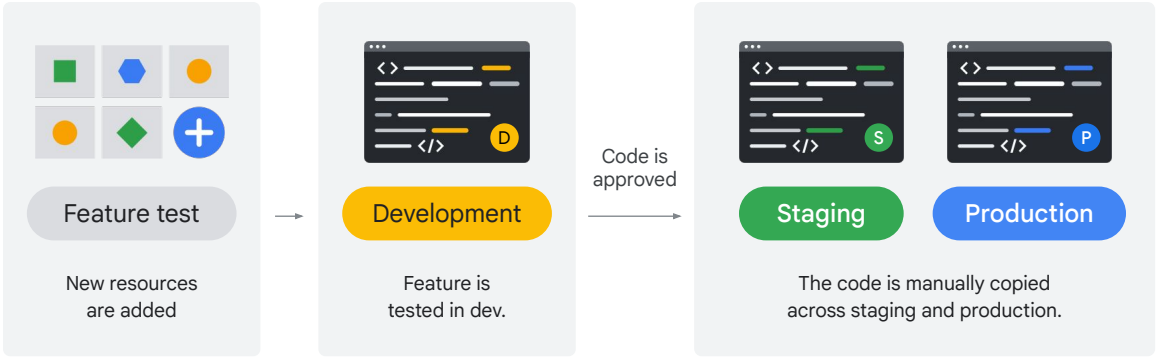
Proprietary + Confidential

Topics

01	Need for modules
02	Modules overview
03	Example of a module, use cases and benefits
04	Reuse configurations by using modules
05	Use variables to parameterize a module
06	Pass resource attribute outside the module by using output values
07	Modules best practices
08	A real time scenario

Proprietary + Confidential

Managing complex infrastructure in Terraform



Google Cloud

Now that you know what modules are, let's look at a simple scenario where they are useful.

In a typical application development environment, when a new feature is added or if an existing feature is modified, the standard workflow to approve a code is that it passes from development to staging and then to production environments. Each environments has the same type of resources but differ in quantity. Let's assume that the current directory structure for this application includes development, staging, and production folders. Thus, the developer will add the code involved in creating the required resource to the development environment first.

Once the feature is approved and tested, without modules, the code is manually copied to staging and then to production.

Proprietary + Confidential

Define all resources that belong to the server in a .tf file

```
-- servers/
   -- main.tf
-- environments/
   -- development/
      -- main.tf
   -- production/
      -- main.tf
   -- staging/
      -- main.tf
```

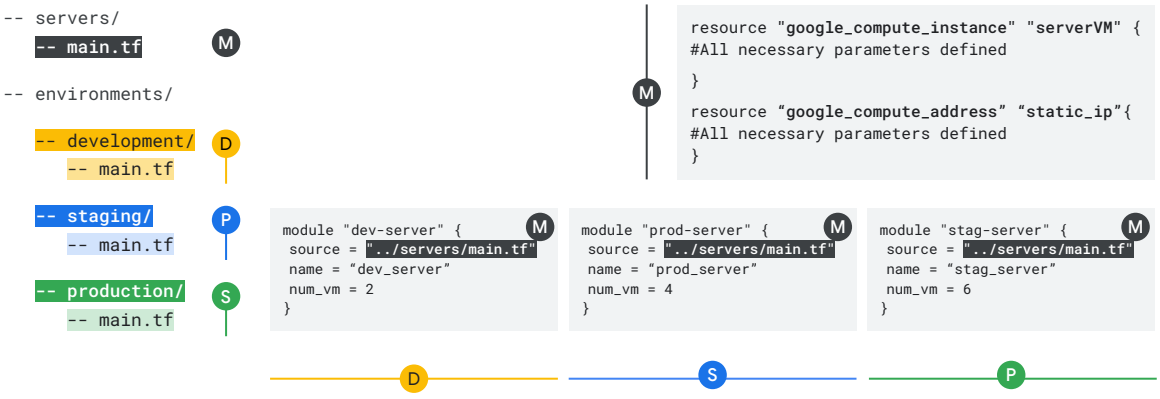
```
resource "google_compute_instance" "server" {
#All necessary parameters defined
}
resource “google_compute_address” “static_ip”{
#All necessary parameters defined
}
```

Google Cloud

Define the reusable code within a module named server. Therefore, any change you make to the module is reflected across all the environments that you plan to reuse.

Proprietary + Confidential

Reuse configuration with modules



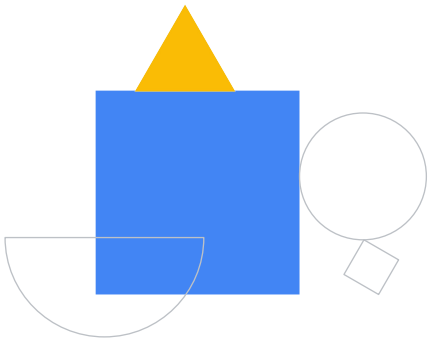
Google Cloud

You can now reference the code without having to make the changes manually across all the environments. In the example shown, the development, staging and production environment can reuse the same module but have different names and deploy different number of servers.

Proprietary + Confidential

Lab

Automating the Deployment of Infrastructure using Terraform



Google Cloud

In this lab, you learn how to perform the following tasks:

- Create a configuration for an auto mode network
- Create a configuration for a firewall rule
- Create a module for VM instances
- Create and deploy a configuration
- Verify the deployment of a configuration

Proprietary + Confidential

Quiz

Proprietary + Confidential

Quiz | Question 1

Question

What is the purpose of output values within modules?

A. Pass resource attributes outside a module

B. Parameterize a configuration

C. Ensure the syntax is in canonical format

D. Initialize Terraform to download the plugins.

Proprietary + Confidential

Quiz | Question 2

Question

Which code construct of Terraform helps you parameterize a configuration

A. Variables

B. Modules

C. Output values

D. Resources

Proprietary + Confidential

Quiz | Question 3

Question

State true or false.
The source of a module can only be remote.

A. True

B. False

Proprietary + Confidential

Quiz | Question 4

Question

What happens when a version argument is specified in a module block?

A. Terraform automatically downgrades the modules to the specific version.

B. Terraform automatically upgrades the modules to the specific version.

C. Terraform automatically upgrades the module to the latest version matching the specified version constraint.

D. Terraform automatically downgrades the module to the oldest version.

Proprietary + Confidential

Module Review

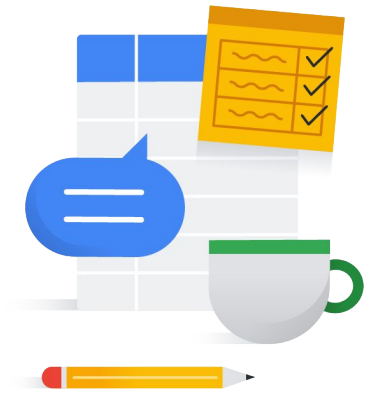
01 Define modules.

02 Use modules to reuse configurations.

03 Use modules from the public registry.

04 Use input variables to parameterize configurations.

05 Use output values to access resource attributes outside the module.



Google Cloud

You learned the definition of a **module**, viewed a few examples, and learned how you can use them to reuse configurations. In addition, you learned how to use input variables to parameterize configurations, and output values to access resource attributes outside the module. This module concluded with best practices and use cases for modules.

Proprietary + Confidential