Google Cloud

Writing Infrastructure Code for Google Cloud

Objectives

Upon completion of this module, you will be able to:

01	Declare the resources within Terraform.
02	Explain implicit and explicit resource dependencies.
03	Use variables and output values within the root configuration.
04	Explain Terraform Registry and Cloud Foundation Toolkit.

Google Cloud

Welcome to the "Writing Infrastructure Code for Google Cloud" module. In this module, you will learn more about resources, variables, and output resources. We will begin by exploring how to create infrastructure components using resources and then explore how Terraform handles dependencies within resources. Although we have been covering resource creation by using hardcoded resource arguments, we will explore how you can parameterize a given configuration using variables. We will explore the syntax to declare, define, and use variables within your configuration. We will then discuss how you can export resource attributes outside the resource declaration by using output values. We will then conclude the module by discovering how you can simplify code authoring by using Terraform registry and Cloud Foundation Toolkit.
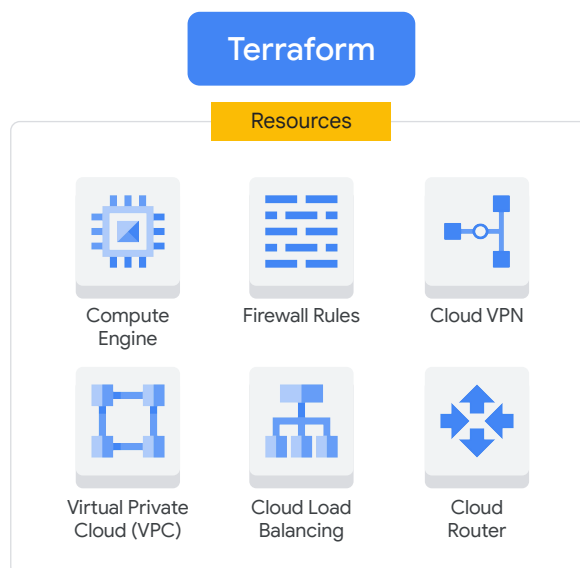Let's begin!

Topics

Google Cloud

What are resources?

- Resources are infrastructure elements you can configure using Terraform.

  Examples:
  Compute Engine instance, VPC, Cloud Storage bucket, Firewall rules

- Terraform uses the underlying APIs of each Google Cloud service to deploy your resources.

Terraform

Resources

| Compute Engine | Firewall Rules | Cloud VPN |
| Virtual Private Cloud (VPC) | Cloud Load Balancing | Cloud Router |

Google Cloud

Resources in the Terraform world are infrastructure elements, such as Compute Engine instances and Cloud Storage buckets, that you can configure with Terraform code. Real-world infrastructure has a diverse set of resources and resource types.

Terraform uses the underlying APIs of each Google Cloud service to deploy your resources. The access to APIs enables you to deploy almost everything we have seen so far, from instances, instance templates, and groups, to VPC networks, firewall rules, VPN tunnels, Cloud Routers, and load balancers.

Syntax to declare a resource

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

```
resource "resource_type" "resource_name" {
  #Resource arguments
}
```

- Resources are defined within a .tf file.
- The resource block represents a single infrastructure object.
- The `resource type` identifies the type of resource being created.
- The `resource type` depends on the provider being declared within a terraform module.
- Not all resource arguments must be defined.

Google Cloud

Let's next explore where and how can you define a resource.

Where?

You can define resources within a .tf file. Terraform recognizes files ending with a .tf extension as configuration files and will load them when it runs. We recommend that you follow a directory structure where resources of similar types are placed within a directory and resources are defined within the main.tf. file. In the slide example, we define the resources within the main.tf file under the network directory. Currently the main.tf is our root configuration. We will explore the other files within the directory later in this module.

How?

The resource block is used to declare an infrastructure object. The resource block represents a single infrastructure object. The resource type identifies the type of resource being created.

The resource type depends on the provider being declared within a Terraform module. A provider is a plugin that provides a collection of resource types. Generally, a provider is a cloud infrastructure platform. In this course, we discuss Google Cloud as the provider.

The resource arguments use expressions to declare the resource attributes. Some

resource arguments are mandatory for resource creation, and a few are optional.

The resource attribute can be used to define any advanced feature associate with a resource.

Refer to the [Terraform Registry](#) for Google Provider for details on resource types and arguments required to be used.

You can include multiple resources of the same type or multiple resources of different resource types within the same Terraform configuration file. These resources can even span across multiple providers. The slide shows an example of two resource blocks. One block that declares the VPC network and another that declares the subnet are placed in the same main.tf file. You can even have multiple resource blocks to define other subnets in the VPC network within the same .tf file.

Examples of a resource block

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

```
resource "google_compute_network" "vpc_network" {
  name                    = "vpc-network"  #Required argument
  project                 = "<Project_ID>"
  auto_create_subnetworks = false
  mtu                     = 1460
}
```

```
resource "google_compute_subnetwork" "subnetwork-ipv6" {
  name                    = "ipv6-test-subnetwork"
  ip_cidr_range           = "10.0.0.0/22"
  region                  = "us-west2"
  network                 = google_compute_network.vpc_network.id
}
```

Google Cloud

As a starting point, let's examine a basic declaration of the resources. Shown on the slide is an example of a VPC using the google_compute_network named vpc_network. A resource is identified by the 'resource' keyword followed by the resource type, in this case google_compute_network. The name is a required resource argument for the google_compute_network block, but the other arguments are optional. You can even define advanced features such as versioning within the same block.

Another example on the slide is a simple definition of a VPC subnet defined by using the google_compute_subnetwork named subnetwork-ipv6. For the subnetwork block, the name, the network in which the subnet should be created, and the IP CIDR range are required arguments that the resource cannot be created without.

Notice that the resource arguments defined within the resource block are dependent on the resource type. This means that a google_compute_network includes arguments such as name, project, and auto_create subnetworks, whereas a google_compute_subnetwork resource type includes arguments such as name, IP_cidr_range, and network.

Syntax for referring to a resource attribute

```
-- main.tf        resource "google_compute_network" "vpc_network" {
                    project                = "my-project-name"
                    name                   = "vpc-network"
                    auto_create_subnetworks = false
                    mtu                    = 1460
                  }

                  resource "google_compute_subnetwork" "subnetwork-ipv6" {
                    name              = "ipv6-test-subnetwork" #Required argument
                    ip_cidr_range     = "10.0.0.0/22" #Required argument
                    region            = "us-west2"
                    network           = google_compute_network.vpc_network.id
                  }
```

- Use **<resource_type>.<resource_name>.<attribute>** to access any resource attributes.
- The Network ID is a computed resource attribute of a **google_compute_network** block.

Google Cloud

Whenever you have to access a resource attribute from another resource block, use the format `<resource_type>.<resource_name>.<attribute>` . In this slide, when a subnet is created, the subnet requires the network ID of the VPC network it belongs to. The network ID is a computed resource attribute that is generated when the network is created. Other computed resource attributes are self_link and IPv4 gateway address. The subnet resource block name subnetwork-ipv6 in the example uses the `network_id` created from the vpc_network block and uses the format `google_compute_network.vpc_network.id.`

Notice that this method can be used only when resources are defined within the same root configuration.
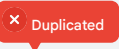
Topics

Google Cloud

Considerations for defining a resource block

- The resource name of a given resource type must be unique within the module.

A resource with the same name cannot exist within the same configuration.

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

```
resource "google_storage_bucket" "dev_bucket" {
  name     = "<unique_bucket_name>"
  location = "US"
}                              ✕ Duplicated

resource "google_storage_bucket" "dev_bucket" {
  name     = "<unique_bucket_name>"
  location = "US"
}
```

Google Cloud

Shown on the slide are some of the rules involved in declaring a resource within the root configuration:

- A declared resource is identified by its resource type and the name; thus the resource name for a given resource type must be unique within the module.

Considerations for defining a resource block

- The resource name of a given resource type must be unique within the module.
- The resource type is not user-defined and is based on the provider.

```
resource "cloud_storage_bucket" "dev_bucket" {
  name          = "<unique_bucket_name>"
  location      = "US"
}
```

User-defined names are not allowed.

```
resource "google_storage_bucket" "dev_bucket" {
  name          = "<unique_bucket_name>"
  location      = "US"
}
```

The resource type is a keyword and must match the term mentioned in Terraform Registry.

Google Cloud

- The resource type is a keyword associated with the provider and thus cannot be user-defined. This means that a Cloud Storage bucket is associated with the Google Cloud provider and is defined by the keyword "google_storage_bucket"; thus it cannot be changed by the user. Any user-defined type would result in an error when terraform plan or terraform apply is executed.

Considerations for defining a resource block

- The resource name of a given resource type must be unique within the module.
- The resource type is not user-defined and is based on the provider.
- All configuration arguments must be enclosed within the resource block.

```
resource "google_storage_bucket" "dev_bucket" {
    name          = "<unique_bucket_name>"
}
    location      = "US"
```

❌

```
Error: Unsupported argument

    On main.tf line 4:
    4: location      = "US"

An argument named "location" is not expected here.
```

Google Cloud

- All configuration arguments must be enclosed within the resource block body, which is between the curly brackets.

Considerations for defining a resource block

- The resource name of a given resource type must be unique within the module.
- The resource type is not user-defined and is based on the provider.
- All configuration arguments must be enclosed within the resource block.
- All required resource arguments must be defined.

```
resource "google_storage_bucket" "dev_bucket" {
location    = "US" #The name argument is missing
}
```
❌

```
resource "google_storage_bucket" "dev_bucket" {
name        = "<unique_bucket_name>"
location    = "US"
}
```
✓

```
Error: Missing required argument

  On main.tf line 9, in resource "google_storage_bucket"
  "mybucket1":
  9: resource "google_storage_bucket" "dev_bucket" {

The argument "name" is required, but no definition was
found.
```

Google Cloud

- An infrastructure element and its associated attributes are defined within a resource block. Some attributes are optional, while some are required. A Terraform configuration will not pass through the plan and apply phases successfully without all the required arguments defined within the configuration.

Topics

Google Cloud

Let's next explore how you can parameterize a configuration by using variables. In this section we will explore how to define and use a variable.

Variables overview

- Variables parameterize your configuration without altering the source code.
- Variables allow you to assign a value to the resource attribute at run time.
- Variables separate source code from value assignments.

```
resource "google_storage_bucket" "mybucket1" {
   name            = "my-project-name" #Required argument
   location        = "US"
   storage_class   = "standard"
}
```

Without variables, resource arguments are hardcoded within the configuration.

Google Cloud

So far we have been hardcoding our values to the various resource arguments.

What if you want to parameterize your configuration and define its value during `terraform apply`?
What if you want to standardize the code but also have the flexibility to customize a few resource attributes at run time?

Variables help you achieve that.

Variables are essential to parameterize values shared between resources (for example, region or zone). Input variables serve as parameters for Terraform, allowing easy customization and sharing without having to alter the source code. After a variable is defined, there are different ways to set its values at run time: environment variables, CLI options, or key-value files.

In the code snippet on the slide, notice that the name, location, and storage class values are hardcoded within the configuration. If you want to be able to decide the name or location or even a storage class of bucket at run time or be able to separate source code from value assignment, you can declare that attribute as a variable. Let's explore how you can do that in the next section.

Syntax to declare an input variable

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

Syntax

```
variable "variable_name" {
  type        = <variable_type>
  description = "<variable description>"
  default     = "<default value for variable>"
  sensitive   = true
}
```

Example: Variable for a bucket region. The variable name must be unique within a module.

```
variable "bucket_region" {
  type        = string
  description = "Region for the bucket"
  default     = "US"
  sensitive   = true
}
```

Google Cloud

You must declare a variable in the variable block. It's recommended to save all variable declarations within a separate file named variables.tf.

The label next to the keyword "variable" give the variables a name. The two rules involved in naming variables that are also common with other languages are:

- The name of the variable must be unique within a module.
- Variable names cannot be keywords.

There are no required arguments for a variable; thus, a variable block can even be empty. The type and default values are not mandatory, and Terraform can automatically deduce the type from the default values assigned to the variable.

Variable arguments: `type`

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

```
variable "variable_name" {
  #Arguments
  type        = <variable_type>
  default     = "<default value for variable>"
  description = "<variable description>"
  sensitive   = true
}
```

```
variable "bucket_region" {
  type        = string
}
```

Google Cloud

The "`type`" argument specifies value types that are accepted for the variable.
Terraform supports the following primitive variable types:

- Bool is used for binary values such as true or false without the double quotes.
- Number is used for numeric variables.
- String is used for a sequence of unicode characters.

Variable arguments: `default`

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

When no value is set for the variable, the value specified in the
`default` argument is used.

```
$terraform plan
Terraform will perform the following actions:

 # google_storage_bucket.mybucket1 will be created
  + resource "google_storage_bucket" "mybucket1" {
      + location                = "US"
      + name                    = "<unique_bucket_name>"
      + project                 = (known after apply)
      + storage_class           = "REGIONAL"
..

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

```
m   resource "google_storage_bucket" "mybucket1" {
    name         = "<unique_bucket_name>"
    location     =  "US"
    storage_class =  var.bucket_storage_class
    //No value assigned for the storage class
    }
```

```
v   variable "bucket_storage_class" {
    type    = string
    default  = "REGIONAL"
    }
```

Google Cloud

The default is another meta argument used to assign a default value to an attribute. The value mentioned in the default argument is used when no value is set for the variable.
To access the value of a variable declared within the module, you can use within the expressions `var.<variable_name>`.

In the example on the slide, the variable `bucket_storage_class` is used in the resource block using the format `var.bucket_storage_class`. Notice that the name of the variable in the variable block has to match the reference made within the resource block.

The value mentioned in default can be overridden by assigning a value in environment values, or .tfvars files or -var option.

There are four items on the slide. The folder structure shows that the variable declaration is written within the variables.tf file, which is a recommended practice. The code next to the folder structure displays the code to assign a variable to storage_class argument. The code in the variables.tf file is the variable definition with the default value specified within quotes. The screenshot below the folder structure shows that the default value is used when creating the resource.

Variable arguments: `description`

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

```
variable "variable_name" {
    type        = <variable_type>
    default     = "<default value>"
    description = "<variable description>"
    sensitive   = true
}
```

```
$terraform plan
var.bucket_region
  Specify the bucket region.

  Enter a value:
```

```
variable "bucket_region" {
    type        = string
    description = "Specify the bucket region."
}
```

The description will be displayed at run time, when no value is assigned for the variable.

Google Cloud

The description is used to document the purpose of the variable. The description is displayed during Terraform apply when no value is assigned for the variable. The description should concisely explain the purpose of the variable and what kind of value is expected. This description string might be included in documentation about the module, and so it should be written from the perspective of the user of the module rather than its maintainer.

Variable arguments: `sensitive`

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

- The acceptable value for `sensitive` is `true`.
- When set to `true`, the value is marked `sensitive` at run time.

```
variable "variable_name" {
   type        = <variable_type>
   default     = "<default value>"
   description = "<variable description>"
   sensitive   = true
}
```

```
$terraform plan
Terraform will perform the following actions:

  # some_resource.a will be created
  + resource "some_resource" "example_resource" {
      + name    = (sensitive)
      + address = (sensitive)
    }

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

```
variable "user_information" {
   type = object({
       name    = string
       address = string
     })
   sensitive = true
}
resource "some_resource" "foo" {
   name    = var.user_information.name
   address = var.user_information.address
}
```

Google Cloud

Sensitive, as the name suggests, is a variable argument used to protect sensitive information from being displayed in command output or log files.

- The acceptable value for `sensitive` is true.
- When set to true, the value is marked sensitive in the output of terraform plan or terraform apply.

This argument is beneficial when dealing with information such as db password, username, or API tokens, and so on,. Marking the variable as sensitive will also eliminate the accidental exposure of confidential information.

In the slide example, the variable called `user_information` is marked sensitive. A resource, for example, foo, uses the sensitive variables, name and address, declared in the `user_information`. When terraform plan or apply is run, observe that the output avoids displaying the values of the variable by marking it as sensitive.

Syntax to reference and assign a value to a variable
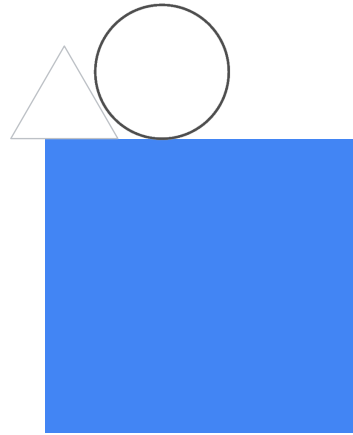
Various ways to assign values to variables

- .tfvars files: Useful for quickly switching between sets of variables and versioning them.
- CLI options: Useful when running quick examples on simple files.
- Environment variables: Useful in scripts and pipelines.
- CLI prompt: If a required variable has not been set via one of the above.

```
# .tfvars file (Recommended method)
$terraform apply -var-file my-vars.tfvars

# CLI options
$terraform apply -var project_id="my-project"

# environment variables
$TF_VAR_project_id="my-project" \
  terraform apply

# If using terraform.tfvars
$terraform apply
```

Google Cloud

There are different ways of setting variable values at run time:

- Use .tfvars files to quickly switch between sets of variables and also be able to version them.
- CLI options are useful when you are running quick examples on simple files.
- Environment variables are useful in scripts and pipelines.
- Use CLI prompt if a required variable has not been set by using one of the other methods.

Assign a value to the variable

Using the terraform.tfvars file

```
-- server/
   -- main.tf
   -- outputs.tf
   -- terraform.tfvars     t
   -- variables.tf         v
```

```
 variable "mybucket_storage_class" {
     type        = string
 }

 variable "bucket_region" {
     type = string
 }
```

```
mybucket_storage_class = "REGIONAL"
bucket_region = "US"
```

Recommended method

Google Cloud

When you have many variable definitions to be inputted to Terraform, providing the value in the command line might not be feasible. The recommended way to assign values to variables is to specify them in a variable definitions file with either a .tfvars or .tfvars.json extension. The variable definitions follow the same syntax as the terraform language but includes only variable name assignments. Whenever files named exactly terraform.tfvars, terraform.tfvars.json, .auto.tfvars, or .auto.tfvars.json are present, Terraform automatically loads the variable definitions from this file.

The definition provided in the .tfvars file overrides the definition in the default argument and environment variable.

Assign a value to the variable

Using the `-var` option

```
-- server/
    -- main.tf
    -- outputs.tf
    -- my-vars.tf      M
    -- terraform.tfvars  T
    -- variables.tf
```

T
```
mybucket_storage_class = "COLDLINE"
bucket_region = "US"
```

M
```
mybucket_storage_class = "NEARLINE"
bucket_region = "EU"
```

```
$cd /server
$terraform apply -var="mybucket_storage_class=REGIONAL"

$terraform apply -var-file my-vars
```

Google Cloud

If you want to specify the value of a variable individually on the command line, you can use the `-var` option as shown on the slide. You can enter these values while running either the terraform plan or terraform apply command.

This method is usually used when running in automation where the -var is sourced from another env variable. It's also useful when running quick examples on simple files.

If you name the files differently with a .tf extension, then you have to specify the filename using the `-var-file` option on the command line. Variables set in the file can be overridden at deployment. This lets you reuse the variables file while still being able to customize the configuration at deployment.

This method takes the highest precedence over all other methods in which a variable value is assigned. In other words, when the variable value is assigned using multiple methods, the value defined using the -var option is the value that is assigned to the variable.

Refer to the [Terraform documentation](#) for more information.

Assign a value to the variable

At run time

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

M

```
resource "google_storage_bucket" "mybucket" {
  name         = "student102345688493"
  location     = var.bucket_region
  storage_class = var.mybucket_storageclass
}
```

```
$terraform plan
var.mybucket_storageclass
  Set the storage class to the bucket.
  Enter a value:
```

V

```
variable "mybucket_storageclass" {
    type        = string
    description = "Set the storage class to the bucket."
}
```

Set the value of the storage class when running
`terraform plan.`

Terraform prompts you on the CLI if a required variable has not been set by using any method described previously. In the example on the slide, the variable mybucket_storageclass is not parameterized and has not been assigned a value within a .tfvars file or in the variable block or in any of the methods discussed before. Thus, you are prompted to enter a value during the plan phase.

Validate variable values by using rules

```
variable "mybucket_storageclass" {
  type        = string
  description = "Set the storage class to the bucket."
  validation {
    condition = contains(["STANDARD", "MULTI_REGIONAL", "REGIONAL"], var.storageclass)
    error_message = "Allowed storage classes are STANDARD, MULTI_REGIONAL and REGIONAL."
  }
}
```

```
var.mybucket_storageclass
  Set the storage class to the bucket.
  Enter a value: ZONAL
google_compute_network.vpc_network: Refreshing state...
|
| Error: Invalid value for variable
|
|   on main.tf line 1:
|    1: variable "mybucket_storageclass" {
|
| Allowed storage classes are STANDARD, MULTI_REGIONAL and REGIONAL.
|
| This was checked by the validation rule at main.tf:4,3-13.
```

Validates the value assigned to the variable.

You can even validate the value assigned to a variable by including a validation subblock with the variable block. The validation block includes a condition argument for which the validation rule is assigned. In this example, a function named contains is used to validate whether the value assigned to a storage class is provided in upper case and is one among accepted values.

Topics

Google Cloud

Let's explore some of the best practices to follow when using variables.

Parameterize only when necessary

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

- Only parameterize values that must vary for each instance or environment.
- Changing a variable with a default value is backward-compatible.
- Removing a variable is *not* backward-compatible.

Only parameterize values that must vary for each instance or environment. When deciding whether to expose a variable, ensure that you have a concrete use case for changing that variable. If there's only a small chance that a variable might be needed, don't expose it.

- Changing or adding a variable with a default value is backward-compatible.
- Removing a variable is *not* backward-compatible.

Provide values in a .tfvars file

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

```
mybucket_storage_class = "REGIONAL"
bucket_region = "US"
```
✔

Command-line options are ephemeral in nature and cannot be checked into source control.

```
$cd /server
$terraform apply -var="mybucket_storage_class=REGIONAL"
$terraform apply -var="bucket_region=US"
$terraform apply -var-file my-vars.txt
```
✖

For root modules, provide values to variables by using a .tfvars variables file. For consistency, name variable files terraform.tfvars.

Don't specify variables by alternating between `var-files` or `var='key=val'` command-line options. Command-line options are ephemeral and easy to forget. They cannot be checked in to source control. Using a default variables file is more predictable.

Give descriptive names to variables

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

```
variable "ram_size_gb" {
  type        = number
  description = "RAM size in GB."
}
```
✓

```
variable "ram_size" {
  type        = number
  description = "RAM size in GB."
}
```
✗

Give variables descriptive names that are relevant to their usage or purpose:

- Variables representing numeric values—such as disk sizes or RAM size—must be named with units (like ram_size_gb). Google Cloud APIs don't have standard units, so naming variables with units makes the expected input unit clear for configuration maintainers.
- To simplify conditional logic, give boolean variables positive names (for example, enable_external_access).

Provide meaningful descriptions

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

```
variable "bucket_region" {
  type        = string
  default     = "US"
  description = "Specify the bucket region."
}
```
✓

```
variable "myregion" {
  type        = string
  default     = "US"
  description = "Specify the region."
}
```
✗

Google Cloud

Variables must have descriptions. Descriptions are automatically included in a published module's auto-generated documentation. Descriptions add additional context for new developers that descriptive names cannot provide.

Topics

Google Cloud

Meta-arguments customize the behavior of resources

count	Create multiple instances according to the value assigned to the count.
for_each	Create multiple resource instances as per a set of strings.
depends_on	Specify explicit dependency.
lifecycle	Define life cycle of a resource.
provider	Select a non-default provider configuration.

Google Cloud

The Terraform language defines several meta-arguments, which can be used with any resource type to change the behavior of resources.

- count, to create multiple instances according to the value assigned to the count.
- for_each, to create multiple instances according to a map or set of strings.
- depends_on, for specifying explicit dependency
- lifecycle, for defining the lifecycle of a resource. Some of the actions you can perform using a lifecycle argument are:
    - Prevent destruction of a resource for compliance purposes.
    - Create a resource before destroying the replaced resource; used for high availability.
- Provider, for selecting a non-default provider configuration. You can have multiple configurations for a provider, including default. Use the provider argument to select an alternate configuration of the provider.

We will explore count and for_each in this course and refer to the Terraform documentation for details on how to use other meta-arguments.

count: Multiple resources of the same type

Redundant code ❌

```
resource "google_compute_instance" "dev_VM1" {
  name         = "dev_VM1"
…}

resource "google_compute_instance" "dev_VM2" {
  name         = "dev_VM2"
…}

resource "google_compute_instance" "dev_VM3" {
  name         = "dev_VM3"
…}
```

Creates three instances of
the same type with names:
- dev_VM1
- dev_VM2
- dev_VM3

✔

```
resource "google_compute_instance" "Dev_VM" {
  count        = 3
  name            = "dev_VM${count.index + 1}"
 #other required arguments
…
}
```

The Terraform count meta-argument lets you deploy multiple resources using the same Terraform configuration block. This is useful when you must deploy infrastructure elements like virtual machines that have the same configuration.

A resource block is used to deploy a single resource. The terraform count and terraform `for_each` meta arguments offer ways to deploy a resource block multiple times. Count works by adding a count parameter to a resource block.

Let's look at deploying multiple VM instances. In the resource definition on the slide, each Compute Engine instance is created by writing three `google_compute_instance` resource blocks. We can replace redundant code by adding the count argument at the beginning of the resource definition.

The count argument in the example tells Terraform to create three instances of the same kind. Notice the expression on the second line. count.index, which represents the index number of the current count loop. The count index starts at 0 and increments by 1 for each resource. You include the count index variable in strings using interpolation. When deployed, Terraform names the VM instances `dev_VM1, dev_VM2, and dev_VM3`.

`for_each`: Multiple resources of the same type with distinct values

Redundant code ❌

```
resource "google_compute_instance" "VM1" {
 name     = "dev-us-central1-a"
 location = "us-central1-a"
 ..
resource "google_compute_instance" "VM2" {
 name     = "dev-asia-east1-b"
 location = "asia-east1-b"
 ..
resource "google_compute_instance" "VM3" {
 name     = "dev-europe-west4-a"
 location = "europe-west4-a"
 ..
```

Creates three instances with the names:
- dev-us-central1-a
- dev-asia-east1-b
- europe-west4-a

✓

```
resource "google_compute_instance" "dev_VM" {
for_each = toset( ["us-central1-a", "asia-east1-b", "europe-west4-a"] )
name     = "dev-${each.value}"

zone     = each.value
#other required arguments
}
```

If your instances are almost identical, count is appropriate. If some of their arguments need distinct values that can't be directly derived from an integer, it's safer to use for_each.

The for_each argument can be assigned to a string of values or map. Terraform will create one instance for each member of the string.

Consider a scenario where you need three similar instances configured in three specific zones and also want the names to have zones as prefixes for identification. Instead of writing lengthy repetitive code, you can use the for_each argument to assign specific values.

The example code creates three instances with the name dev-us-central1-a, dev-asia-east1-b, and europe-west4-a in their respective zones.

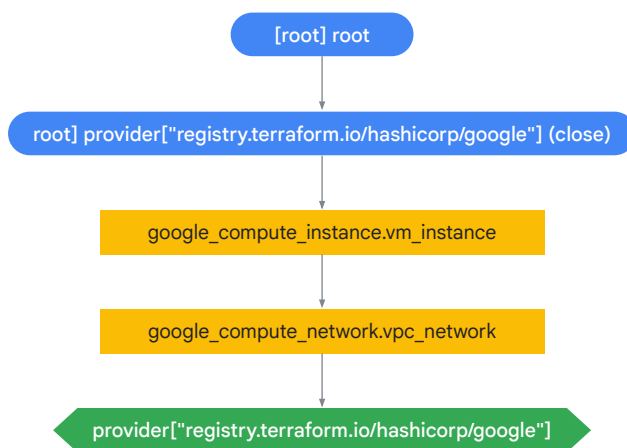Refer to the [Terraform documentation](#) for more information on meta arguments.

Topics

Google Cloud

Dependency graph

- Built from the terraform configurations.
- Interpolates attributes during run time.
- Determines the correct order of operations.

Google Cloud

While building an infrastructure, you may prefer to have a visual representation of how your infrastructure is connected and interdependent. Terraform has a feature called dependency graph that serves that exact purpose. It adds a layer to understand how your infrastructure looks before you actually deploy it. Terraform builds a dependency graph from the Terraform configurations and walks this graph to generate plans and refresh state. The attributes are interpolated during run time, and primitives such as variables, output values, and providers are connected in a dependency tree. When performing operations, Terraform creates a dependency graph to determine the correct order of operations. In more complicated cases with multiple resources, Terraform will perform operations in parallel when it's safe to do so.

Resource dependencies

Terraform can handle two kinds of dependencies.

Implicit dependency	Explicit dependency
Dependencies **known** to Terraform are detected automatically.	Dependencies **unknown** to Terraform must be configured explicitly.

Google Cloud

Terraform can handle two kinds of dependencies between the resources it manages. They are implicit dependency and explicit dependency.

In short, implicit dependency is a dependency known to Terraform, whereas explicit is a dependency that is unknown to Terraform.

Based on the nature of how resources work, there are scenarios where one resource creation depends on the information generated after the creation of another resource. For example, you cannot create a compute instance unless the network is created. Similarly, you cannot assign a static IP address for a Compute Engine instance until a static IP is reserved. Such dependency is called implicit dependency.

There are also scenarios where you induce a dependency. A given resource can only be created upon creation of another resource. In such cases you would want to explicitly mention dependency that Terraform cannot see. For example, perhaps an application you will run on your instance expects to use a specific Cloud Storage bucket, but that dependency is configured inside the application code and thus not visible to Terraform. In that case, you can use `depends_on` to explicitly declare the dependency.

Implicit resource dependencies are handled automatically

```
resource "google_compute_instance" "my_instance" {
  //All mandatory arguments

  network_interface {
   //implicit dependency
    network = google_compute_network.my_network.name
    access_config {
    }
  }
}

resource "google_compute_network" "my_network" {
    name = "my_network"
}
```

The reference to **mynetwork** in the network argument creates an implicit (**known**) dependency.

Google Cloud

Implicit dependencies

Let's see how Terraform handles implicit dependencies with an example. In the example on the slide, the compute instance my_instance is created in a custom VPC called my_network. In Google Cloud, you cannot create a compute instance without a network.

Implicit dependencies by using interpolation expressions are the primary way to inform Terraform about these relationships, and should be used whenever possible. In our example, the reference to my_network in the network argument creates an implicit dependency on the network mentioned in the google_compute_network block.

View implicit (known) dependencies
via `terraform apply`

Terraform creates the network first.

After the implicit dependency is fulfilled (network is created), the compute instance is created.

```
$terraform apply

google_compute_network.mynetwork: Creating...
google_compute_network.mynetwork: Still creating... [10s elapsed]
google_compute_network.mynetwork: Still creating... [20s elapsed]
google_compute_network.mynetwork: Still creating... [30s elapsed]
google_compute_network.mynetwork: Creation complete after 32s
[id=projects/qwiklabs-gcp-01-e973d950dd4a/global/networks/mynetwork]

google_compute_instance.myinstance: Creating...
google_compute_instance.myinstance: Still creating... [10s elapsed]
google_compute_instance.myinstance: Creation complete after 13s
[id=projects/qwiklabs-gcp-01-e973d950dd4a/zones/us-central1-a/instances/myinstance]
```

Snippet of the `terraform apply` output

Google Cloud

You can view the order of resource creation when you execute the `terraform apply` command. Terraform by default knows the order in which the resource must be created. Due to implicit dependency, Terraform is able to infer a dependency and knows it must create the network before creating the instance. In this example, Terraform creates the VPC `my_network` before creating the compute instance named `my_instance`.

When Terraform reads this configuration, it will:

- Ensure that `my_network` is created before my_instance.
- Save the properties of `my_network` in the state.
- Set the network argument in google_compute_instance to the value of the name argument in the `google_compute_network` block.

Explicit (unknown) dependencies are defined by using the depends_on argument

depends_on
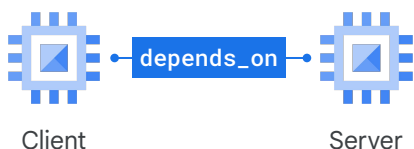
Client   Server

The client VM can only be created
when the server VM is created.

```
resource "resource_type" "resource_name" {

..
depends_on = [<resource_type>.<resource_name>]
}
```

```
resource "google_compute_instance" "client" {
 ...
 depends_on = [google_compute_instance.server]
}

resource "google_compute_instance" "server" {

 #All required configuration options
}
```

Google Cloud

There are dependencies between resources that are not visible to Terraform. This sequence can be enforced using the depends_on argument. We will look at the code snippet in the next slide. You can induce explicit dependency by introducing the depends_on argument within the dependent resource block. The depends_on argument gives you the flexibility to control the order in which Terraform processes the resource with a configuration.

The depends_on argument can be used within the module block regardless of the resource type. The value of the depends_on argument can be an expression that directs to the resource mentioned within the square brackets using the format shown on the slide.
For example, if you want to create two VMs—server and client —and want the client VM to only be created upon successful creation of the server VM, then dependency has to be induced. Such dependency is not visible to Terraform and has to be explicitly mentioned. In that case, you can use depends_on to explicitly declare the dependency of the client VM on the server VM.

You may wonder where in your configuration these resource dependencies should be defined. The order in which the resources are defined in a Terraform configuration file has no effect on how Terraform applies your changes. Organize your configuration files in a way that makes the most sense for you and your team.

View explicit (unknown) dependencies via `terraform apply`

Server is created *before* client.

Due to explicit dependency, the client is created only *after* the server is created.

```
$terraform apply
google_compute_instance.server: Creating...
google_compute_instance.server: Still creating... [10s elapsed]
google_compute_instance.server: Creation complete after 12s
[id=projects/qwiklabs-gcp-01-e973d950dd4a/zones/us-central1-a/instances/server]
google_compute_instance.client: Creating...
google_compute_instance.client: Still creating... [10s elapsed]
google_compute_instance.client: Creation complete after 13s
[id=projects/qwiklabs-gcp-01-e973d950dd4a/zones/us-central1-a/instances/client]
```

Snippet of the **terraform apply** output

Google Cloud

Upon executing `terraform apply`, you will notice that the client VM is created after the server VM. During interpolation of expression, Terraform processes the resource specified in the meta argument `depends_on` and creates the server VM even before creating the client VM.
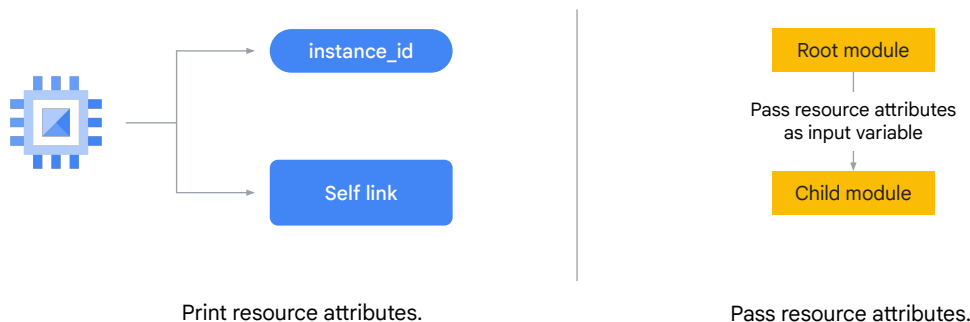
Topics

Google Cloud

What are output values?
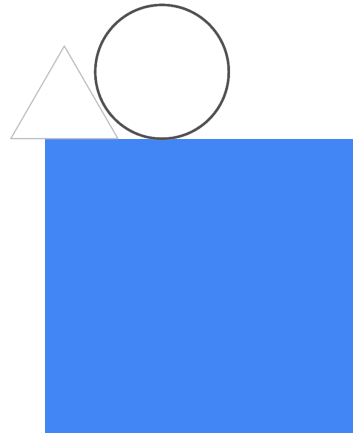
Let's next explore how to print resource attributes using output values. In this section we will explore how to define and use output values.

What are output values?

Output values expose information about the resource to the user of the Terraform configuration.

Print resource attributes.

Pass resource attributes.

Output values are code constructs used for you to view information about the infrastructure resources you created on the command line. You can think of output values as similar to return values in a programming language. They are a way to expose the information about the resource to the user of the Terraform configuration.

Output values are used for several purposes.

- The most common use case of output values is to print some of the resource attribute of a root module in the CLI after its deployment. For example, most of the server details are calculated at deployment and can only be inferred post-creation.
- They are also used to pass information generated by one resource to other. For example, by using output variables, you can extract the server-specific values such as IP address to another resource that requires this information.

We will explore the first use case in this module and the second later in the next module.

Print resource attributes by using output values

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

M

```
resource "google_storage_bucket_object" "picture" {
…
}
```

O

```
output "picture_URL" {
  description = "URL of the picture uploaded
//value         = <resource_type>.<resource_name>.<attribute>
  value        = google_storage_bucket_object.picture.self_link
}
```

```
Google_storage_bucket_object.picture: Creating...
Google_storage_bucket_object.picture: Creating complete after 1s
[]

Apply complete! Resources: 1 added, 0 changed, 0 destroyed.
Outputs:
picture_URL = "https://storage.googleapis.com/my-gallery/..
```

Note: We recommend that you use output values, instead of user-supplied inputs, for computed attributes of a resource.

Google Cloud

You can declare an output value by using the output block. The keyword 'output' is used to indicate to Terraform that the label associated with the keyword is the name of the output value. You can declare an output value anywhere within a configuration, but the recommended practice is to declare them in a separate file called output.tf. The arguments that can be included within an output block are:

- value, which is a required argument that can be an expression whose value is returned to the user of the module.
- description is an optional argument used to provide an explanation of the purpose of the output and the value expected. It can be used for documentation purposes too.
- sensitive is also an optional argument used to mask the value to a resource attribute. This is useful to hide the accidental display of a resource attribute that is meant to be confidential, such as password information.

In the example on the slide, we have declared an output value for the object named 'picture'. After the object is successfully uploaded, the object URL is displayed on the screen.

We recommend that you use output values, instead of user-supplied inputs, for the computed attributes of a resource.

terraform output

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

```
                                    O
```

```
resource "google_compute_network" "vpc_network" {
project= "<PROJECT_ID>"
name = "vpc-network"
}
output network_id {
    value = google_compute_network.vpc_network.id
}
output network_link {
    value = google_compute_network.vpc_network.self_link
}
```

Query all output
values.

```
$terraform output
network_id = "projects/<project-id>/global/networks/vpc-network"
network_link = "https://www.googleapis.com/../projects/<project-id>/../vpc-network"
```

Google Cloud

You can query all the output values used within a project by using the terraform output command. The output of this command lists all the output values used. You can even query individual outputs by name by using the terraform output <output value name> command.

Topics

Google Cloud

Outputs Best Practices
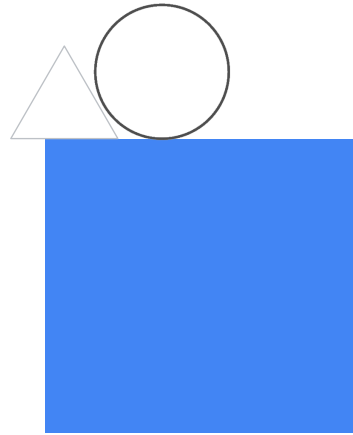
Let us next explore some best practices involved in using output values.

Output useful information

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

```
resource "google_compute_network" "vpc_network" {
    project= "<PROJECT_ID>"
    name = "vpc-network"
}
output network_id {
    value = google_compute_network.vpc_network.id     ✓
}
output network_name {
    value = google_compute_network.vpc_network.name    ✗
}
```

```
Apply complete! Resources: 1 added, 0 changed, 0 destroyed.
Outputs:
network_id = "projects/<project-id>/global/networks/vpc-network"
network_name = "vpc-network"
```

Google Cloud

Output information that is useful to end users. Do not output values that simply regurgitate variables or provide known information to end users. Computed information is helpful to end users. For example, for a network resource, the following computed attributes are exported:

- `id` - an identifier for the resource with format `projects/{{project}}/global/networks/{{name}}`
- `gateway_ipv4` - The gateway address for default routing out of the network. This value is selected by Google Cloud.
- `self_link` - The URI of the created resource.

As in the example shown on the slide, instead of outputting a user inputted value like the network name, we recommend that you output computed attributes that are not visible to the user in the terraform code.

Provide meaningful descriptions

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

Provide meaningful descriptions for all outputs.

```
output "dev_server_URI" {
  description = "URI of the Dev instance"
  value       = google_compute_instance.dev_main.name
}
```

```
output "link" {
  description = "Mylink"
  value       = google_compute_instance.dev_main.name
}
```

Provide meaningful names and descriptions. Descriptions are automatically included in a published module's auto-generated documentation. Descriptions add additional context for new developers that descriptive names cannot provide.

Organize all outputs in an `outputs.tf` file

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

```
output "bucketname" {
  value = google_storage_bucket.mybucket.name
}
output "bucketlocation" {
  value = google_storage_bucket.mybucket.location
}
```
✓

```
resource "google_storage_bucket" "mybucket" {
  name        = "student102345688493"
  location    = "US"
}
output "bucketname" {
  value = google_storage_bucket.mybucket.name
}
output "bucketlocation" {
  value = google_storage_bucket.mybucket.location
}
```
✗

Google Cloud

Organize your code to include all output values in a file named output.tf.

Mark sensitive outputs

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

```
output "sql_user_password" {
  value       = google_sql_user.users.password
  description = "The password for sql user."
  sensitive   = true
}
```

Sensitive values are masked with the keyword "sensitive" in the output of `terraform plan` or `terraform apply`

```
$terraform plan
Terraform will perform the following actions:

  # some_resource.a will be created
  + resource "google_sql_user" "users" {
      + password   = (sensitive)
    }

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

Google Cloud

Mark sensitive outputs: Instead of attempting to manually encrypt sensitive values, rely on Terraform's built-in support for sensitive state management. When exporting sensitive values to output, make sure that the values are marked as sensitive. When a value is marked sensitive, the data is masked from the output of `terraform plan` or `terraform apply` command.

Topics

| 09 | Terraform Registry and Cloud Foundation Toolkit |

The Terraform Registry

- Interactive resource for discovering providers and modules.
- Solutions developed by HashiCorp, third-party vendors, and the Terraform community.

registry.terraform.io/browse/modules?provider=google
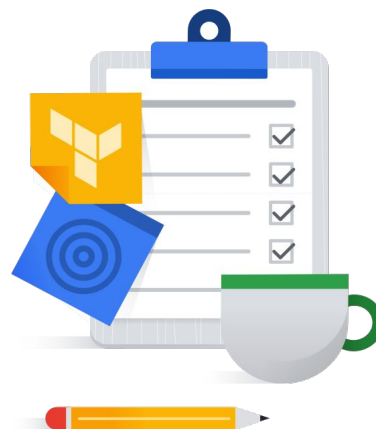
The Terraform Registry is an interactive resource for discovering a wide selection of integrations (providers) and configuration packages (modules) for use with Terraform.

The Registry includes solutions developed by HashiCorp, third-party vendors, and the Terraform community.

The Registry provides plugins to manage any infrastructure API, pre-made modules to quickly configure common infrastructure components, and examples of how to write quality Terraform code.

Cloud Foundation Toolkit (CFT)

- CFT provides a series of reference modules for Terraform that reflect Google Cloud best practices.
- CFT modules can be used without modification to quickly build a repeatable enterprise-ready foundation in Google Cloud.
- CFT modules are also referred to as Terraform blueprints.

cloud.google.com/foundation-toolkit

cloud.google.com/docs/terraform/blueprints/terraform-blueprints

You can also use **Cloud Foundation Fabric (CFF),** a collection of modules and examples for fast prototyping or to be modified and used in organizations.
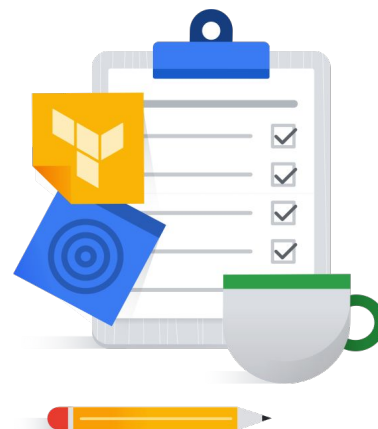
Google Cloud

To take the simplicity of using Terraform one step further, we have the Cloud Foundation Toolkit (CFT). CFT is a library of reusable code in which we implement all of the best practices, thus eliminating the need to relearn all best practices and rebuild them.

You can use this library to automatically adopt our best practices really easily and get that consistent baseline. You can then spend more time focused on the customer.

We also have Cloud Foundation Fabric (CFF), a collection of Terraform modules and end to end examples meant to be cloned as a single unit and used as is for fast prototyping or decomposed and modified for usage in organizations. The repository on github provides end to end blueprints, and a suite of Terraform modules for Google Cloud, which support different use cases.
For more information, see Cloud Foundation Fabric.

The Terraform Registry is the tool for surfacing/distributing modules for all providers (Google, AWS, and so on). CFT is the name of a collection of Google Cloud Terraform modules build and maintained by Googlers. These modules are published to the Terraform Registry.
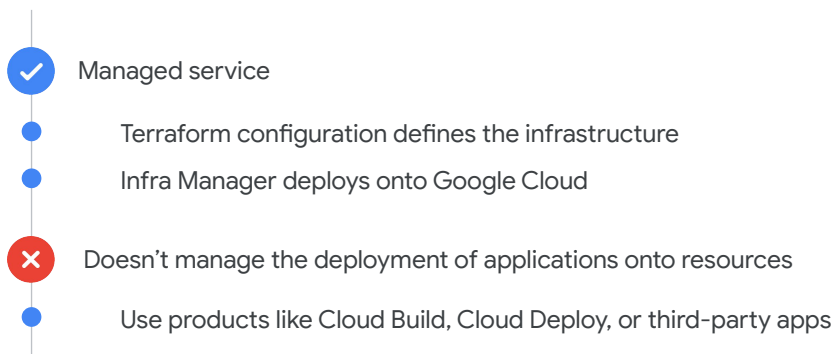
The CFT module lets you maintain the IAM roles for multiple projects within the same module, as opposed to using the standard Terraform where you must update roles for each project individually.

The left side of slide shows a standard Terraform IAM binding to:

- Assign the network user, network group, and the network admin role on project-one and project-two.
- Assign the App Engine user, App Engine group, and the App Engine admin role on project-one and project-two.

The right side of the slide shows the equivalent CFT module where the role and user bindings for multiple projects can be defined within the same module.

Infrastructure Manager

- ✓ Managed service
    - Terraform configuration defines the infrastructure
    - Infra Manager deploys onto Google Cloud
- ✗ Doesn't manage the deployment of applications onto resources
    - Use products like Cloud Build, Cloud Deploy, or third-party apps

Google Cloud

Infrastructure Manager (Infra Manager) is a managed service that automates the deployment and management of Google Cloud infrastructure resources.

Infrastructure is defined using Terraform and deployed onto Google Cloud by Infra Manager, enabling you to manage resources using Infrastructure as Code (IaC).

Infra Manager doesn't manage the deployment of applications onto your resources. To manage application deployment, you can use Google Cloud products like Cloud Build and Cloud Deploy. You can also use third-party tools or your own toolchain.

For more information, see Infrastructure Manager Overview

Lab

Creating Resource
Dependencies

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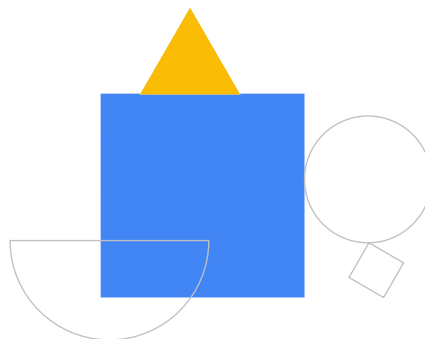
In this lab, you learn how to perform the following tasks:

- Use variables and output values
- Observe implicit dependency
- Create explicit resource dependency

Quiz

Quiz | Question 1

Question

What is the most common use case for output values in Terraform?

A.   Declare a resource within a Terraform configuration.

B.   Parameterize a resource configuration.

C.   Print resource attributes of a root module CLI after its deployment.

Quiz | Question 1

Answer

What is the most common use case for output values in Terraform?

A. Declare a resource within a Terraform configuration.

B. Parameterize a resource configuration.

C. Print resource attributes of a root module CLI after its deployment. ✅

Quiz | Question 2

Question

Can a variable be assigned values in multiple ways?

A.   Yes

B.   No

Quiz | Question 2

Answer

A variable can be assigned values in multiple ways.

A. True ✅

B. False

Quiz | Question 3

Question

Which dependency can be automatically detected by Terraform?

A.   Implicit dependency

B.   Explicit dependency

Quiz | Question 3

Answer

Which dependency can be automatically detected by Terraform?

A. Implicit dependency ✅

B. Explicit dependency

Quiz | Question 4

Question

How many resource types can be represented in a single resource block?

A. Four

B. Three

C. Two

D. One

Quiz | Question 4

Answer

How many resource types can be represented in a single resource block?

A. Four

B. Three

C. Two

D. One ✅

Module review

01	Declare the resources within Terraform.
02	Explain implicit and explicit resource dependencies.
03	Use variables and output values within the root configuration.
04	Explain the Terraform Registry and the Cloud Foundation Toolkit.

Writing infrastructure code is a fundamental skill that you must develop to manage your infrastructure on Google Cloud. In this module, you learned how to declare resources: infrastructure elements you can configure using Terraform. In addition, you examined resource blocks. You also learned how to specify resource dependencies and how and why to use variables and output values within a configuration. This module also explained the Terraform Registry and the Cloud Foundation Toolkit, two tools for simplifying the code writing process.