
Terraform
VARIABLES & OUTPUTS

❖ VARIABLES & OUTPUTS:

- Variables in Terraform are a great way to define **centrally controlled reusable values**.
- If you're familiar with traditional programming languages, it can be useful to compare **Terraform modules to function definitions:**
  - **Input variables** are like function arguments.
  - **Output values** are like function return values.
  - **Local values** are like a function's temporary local variables.

➢ INPUT VARIABLES:

- Input variables let you customize aspects of Terraform modules without altering the module's own source code. This functionality allows you to share modules across different Terraform configurations, making your module composable and reusable.
- Input variables support multiple data types:
  - **Simple Data Types:** String, number, bool are simple data types,
  - **Complex data types:** list, map, tuple, object, and set are
- Each input variable declared using a **variable block**:

SYNTAX: **variable** "variable_name" **{}**

ARGUMENTS:

- Terraform CLI defines the following optional arguments for variable declarations:
  DEFAULT: A default value which then makes the variable optional.
  TYPE: It specifies what value types are accepted for the variable.
  DESCRIPTION: This specifies the input variable's documentation.
  VALIDATION: A block to define validation rules, usually in addition to type constraints.
  SENSITIVE: Limits Terraform UI output when the variable is used in configuration.
  NULLABLE: Specify if the variable can be null within the module.

DECLARING AN INPUT VARIABLE:

- Each input variable must be declared using a **variable** block:
- Create a file with name **variables.tf**

```
variable "availability_zone_names" {
  type    = list(string)
  default = ["us-east-1"]
}

variable "ami" {
 type        = string
 description = "AMI ID for the EC2 instance"
 default     = "ami-079db87dc4c10ac91"

 validation {
   condition  = length(var.ami) > 4 && substr(var.ami, 0, 4) == "ami-"
   error_message = "Please provide a valid value for variable AMI."
 }
}

variable "type" {
 type        = string
 description = "Instance type for the EC2 instance"
 default     = "t2.micro"
 sensitive   = true   }
```

```
variable "tags" {
 type = object({
   name = string
   env  = string
 })
 description = "Tags for the EC2 instance"
 default = {
   name = "My Virtual Machine"
   env  = "Dev"
 }
}

variable "subnet" {
 type        = string
 description = "Subnet ID for network interface"
 default     = "subnet-0f0f0f90b5e13eeea"
}
```

USING INPUT VARIABLE VALUES:

- Within the module that declared a variable, its value can be accessed from within expressions as **var.<NAME>,** where **<NAME>** matches the label given in the declaration block:

  NOTE: Input variables are created by a variable block, but you reference them as attributes on an object named **var.**

CREATE A FILE WITH NAME MAIN.TF**:**

```
resource "aws_instance" "myvm" {
availability_zone  = var.availability_zone_names
ami           = var.ami
 instance_type = var.type
 tags          = var.tags
}
```

```
$Terraform fmt
$terraform validate
$terraform plan
$terraform apply
$terraform destroy
```

ASSIGNING VALUES TO ROOT MODULE VARIABLES:

- When variables are declared in the root module of your configuration, they can be set in a number of ways:
  - Individually, with the **-var** command line option.
  - In variable definitions **(.tfvars)** files, either specified on the command line or automatically loaded.
  - As environment variables.

VARIABLES ON THE COMMAND LINE (-VAR):

- To specify individual variables on the command line, use the -var option when running the terraform plan and terraform apply commands:

  $terraform apply **-var**="image_id=ami-abc123"

  $terraform apply **-var**='image_id_list=["ami-abc123","ami-def456"]' -var="instance_type=t2.micro"

  $terraform apply **-var**='image_id_map={"us-east-1":"ami-abc123","us-east-2":"ami-def456"}'

VARIABLE DEFINITIONS (.TFVARS) FILES:

- To set lots of variables, it is more convenient to specify their values in a variable definitions file (with a filename ending in either **.tfvars** or **.tfvars.json**) and then specify that file on the command line with **-var-file**:

  #terraform apply **-var-file**="testing.tfvars"

  #terraform apply **-var-file**='testing.tfvars'

  #terraform apply **-var-file**="testing.tfvars"

- Terraform also automatically loads a number of variable definitions files if they are present:

  Files named exactly **terraform.tfvars** or **terraform.tfvars.json**.

  Any files with names ending in **.auto.tfvars** or **.auto.tfvars.json**.

ENVIRONMENT VARIABLES:

- As a fallback for the other ways of defining variables, Terraform searches the environment of its own process for environment variables named TF_VAR_ followed by the name of a declared variable.
- This can be useful when running Terraform in automation, or when running a sequence of Terraform commands in succession with the same variables. For example, at a bash prompt on a Unix system:

  #**export TF_VAR_**image_id=ami-abc123

  #terraform plan

VARIABLE DEFINITION PRECEDENCE:

- The above mechanisms for setting variables can be used together in any combination. If the same variable is assigned multiple values, Terraform uses the last value it finds, overriding any previous values. Note that the same variable cannot be assigned multiple values within a single source.
- Terraform loads variables in the following order, with later sources taking precedence over earlier ones:
  - Environment variables
  - The terraform.tfvars file, if present.

- The terraform.tfvars.json file, if present.
- Any *.auto.tfvars or *.auto.tfvars.json files, processed in lexical order of their filenames.
- Any -var and -var-file options on the command line, in the order they are provided. (This includes variables set by a Terraform Cloud workspace.)

➢ OUTPUT VALUES:

- Output values make information about your infrastructure available on the command line, and can expose information for other Terraform configurations to use.
- Output values are similar to return values in programming languages.
- Output values have several uses:
  - A child module can use outputs to expose a subset of its resource attributes to a parent module.
  - A root module can use outputs to print certain values in the CLI output after running terraform apply.
  - When using remote state, root module outputs can be accessed by other configurations via a terraform_remote_state data source.

DECLARING AN OUTPUT VALUE:

- Each output value exported must be declared using an **output** block:

```
resource "aws_instance" "myvm" {
  ami = "ami-079db87dc4c10ac91"
  instance_type = "t2.micro"
  tags = {
        Name = "My EC2"
        Env  = "Test"
        }
      }
```

```
output "PublicIPAddress" {
 value       = aws_instance.myvm.public_ip
 description = "AWS EC2 instancce Public IP"
}

output "PrivateIPAddress" {
 value       = aws_instance.myvm.private_ip
 description = "AWS EC2 instancce Private IP"
}

output "InstanceState" {
 value       = aws_instance.myvm.instance_state
 description = "AWS EC2 instancce State"
}
```

SENSITIVE: Suppressing Values in CLI Output.

- An output can be marked as containing sensitive material using the optional sensitive argument:

```
output "PrivateIPAddress" {
 value       = aws_instance.myvm.private_ip
 description = "AWS EC2 instancce Private IP"
 sensitive = true
}
```

➢ LOCAL VALUES:

- A local value assigns a name to an expression, so you can use the name multiple times within a module instead of repeating the expression.
- Local values are like a function's temporary local variables.

DECLARING A LOCAL VALUE:

- A set of related local values can be declared together in a single locals block:

```
locals {
 ami  = "ami-079db87dc4c10ac91"
 type = "t2.micro"
 subnet_group = "subnet-0f0f0f90b5e13eeea"
 security_id = ["sg-04cecc7e117da949e"]
 keyname = "ram"
 tags = {
   Name = "My_Server"
   Env  = "Prod"
 }
}
```

```
locals {
  # Common tags to be assigned to all resources
  common_tags = {
    Service = local.service_name
    Owner   = local.owner
  }
}
```

USING LOCAL VALUES:

```
resource "aws_instance" "myvm" {
 ami           = local.ami
 instance_type = local.type
 tags          = local.tags
 subnet_id     = local.subnet_group
 key_name      = local.keyname
 }
```

NOTE: Local values are created by a **locals block**, but you reference them as attributes on an object named **local**.

WHEN TO USE LOCAL VALUES:

- Local values can be helpful to avoid repeating the same values or expressions multiple times in a configuration, but if overused they can also make a configuration hard to read by future maintainers by hiding the actual values used.
- Use local values only in moderation, in situations where a single value or result is used in many places and that value is likely to be changed in future. The ability to easily change the value in a central place is the key advantage of local values.