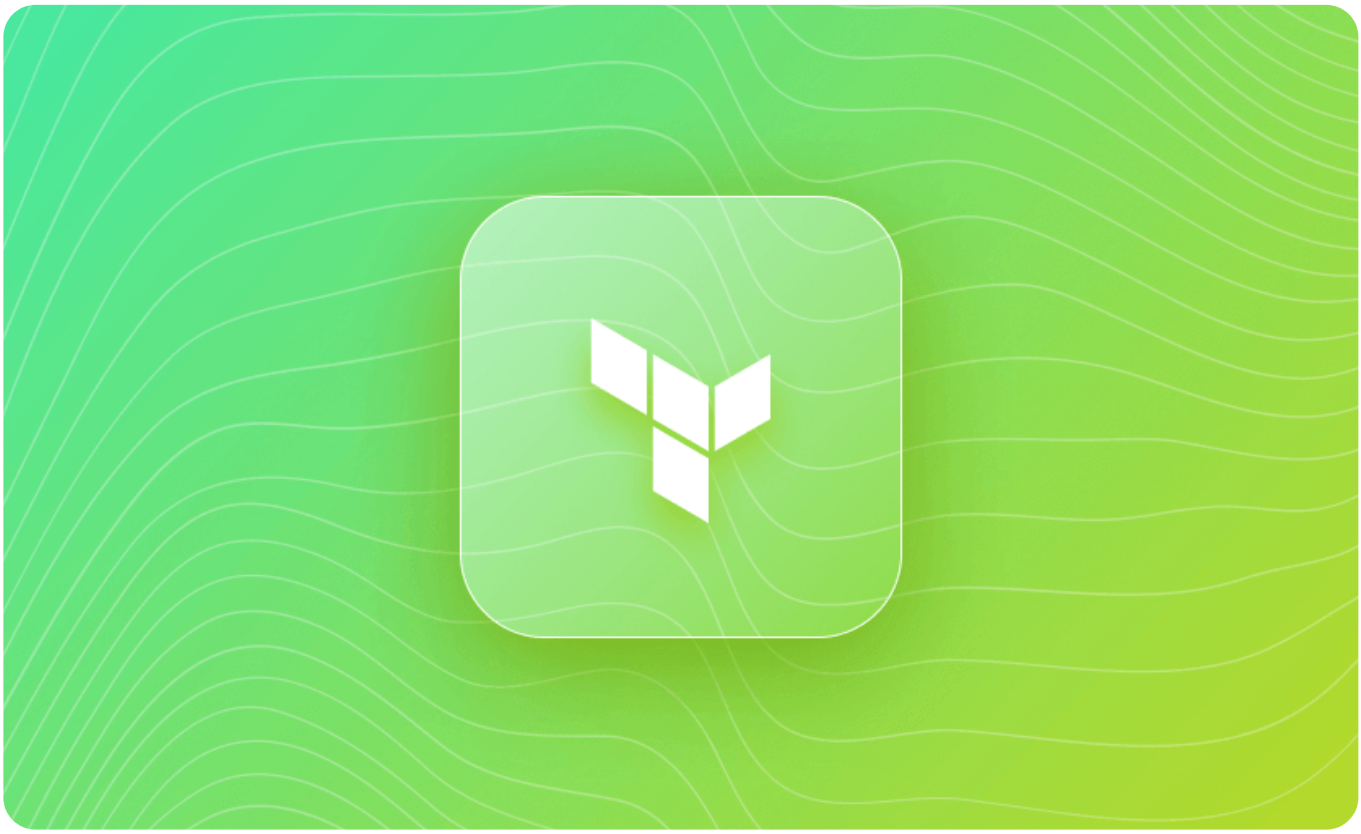
TERRAFORM

How to Use Terraform Variables (Locals, Input, Output, Environment)

Sumeet Ninawe

10 Aug 2021 · 13 min read



Variables are fundamental constructs in every programming language because they are inherently useful in building dynamic programs. We use variables to store temporary values so that they can assist programming logic in simple as well as complex programs.

In this post, we discuss how variables are used in Terraform. Terraform uses HCL (Hashicorp Configuration Language) to encode infrastructure. It is declarative in nature, meaning several blocks of code are declared to create a desired set of infrastructure.

Before we dive into various types of variables, it is helpful to think of the complete Terraform configuration as a single function. As far as variables are concerned, we use

Types of Variables

Local Variables

Local variables are declared using the `locals` block. It is a group of key-value pairs that can be used in the configuration. The values can be hard-coded or be a reference to another variable or resource.

Local variables are accessible within the module/configuration where they are declared. Let us take an example of creating a configuration for an EC2 instance using local variables. Add this to a file named `main.tf`.

```
locals {
 ami  = "ami-0d26eb3972b7f8c96"
 type = "t2.micro"
 tags = {
   Name = "My Virtual Machine"
   Env  = "Dev"
 }
 subnet = "subnet-76a8163a"
 nic    = aws_network_interface.my_nic.id
}

resource "aws_instance" "myvm" {
 ami           = local.ami
 instance_type = local.type
 tags          = local.tags

 network_interface {
   network_interface_id = aws_network_interface.my_nic.id
   device_index         = 0
 }
}
```

```
  description = "My NIC"
  subnet_id   = var.subnet

  tags = {
    Name = "My NIC"
  }
}
```

In this example, we have declared all the local variables in the locals block. The variables represent the AMI ID (`ami`), Instance type (`type`), Subnet Id (`subnet`), Network Interface (`nic`) and Tags (`tags`) to be assigned for the given EC2 instance.

In the `aws_instance` resource block, we used these variables to provide appropriate values required for the given attribute. Notice how the local variables are being referenced using a `local` keyword (without 's').

Usage of local variables is similar to data sources. However, they have a completely different purpose. Data sources fetch valid values from the cloud provider based on the query filters we provide. Whereas we can set our desired values in local variables — they are **not dependent on the cloud providers.**

It is indeed possible to assign a value from a data source to a local variable. Similarly to how we have done it to create the `nic` local variable, it refers to the `id` argument in the `aws_network_interface` resource block.

As a best practice, try to keep the number of local variables to a minimum. Using many local variables can make the code hard to read.

If you want to know more about locals, see: Terraform Locals: What Are They, How to Use Them, Examples

Input Variables

Input variables are similar to local variables. They are **used to assign dynamic values to resource attributes**. However, the difference is in the way they are used. Input variables allow you to pass values before the code execution.

Further, the main function of the input variables is to act as inputs to modules. Modules are self-contained pieces of code that perform certain predefined deployment tasks. Input variables declared within modules are used to accept values from the root directory.

Additionally, it is also possible to set certain attributes while declaring input variables, as below:

- `type` — to identify the type of the variable being declared.
- `default` — default value in case the value is not provided explicitly.
- `description` — a description of the variable. This description is also used to generate documentation for the module.
- `validation` — to define validation rules.
- `sensitive` — a boolean value. If true, Terraform masks the variable's value anywhere it displays the variable.

Input variables **support multiple data types**. They are broadly categorized as simple and complex. `String`, `number`, `bool` are simple data types, whereas `list`, `map`, `tuple`, `object`, and `set` are complex data types.

Let us work through the same example as before, only this time we use variables instead of local variables. Create a new file to declare input variables as `variables.tf` and add the below content to it.

```
variable "ami" {
 type        = string
 description = "AMI ID for the EC2 instance"
 default     = "ami-0d26eb3972b7f8c96"
```

```
    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-76a8163a"
}
```

Here we have declared 5 variables — `ami`, `nic`, `subnet` and `type` with the simple data type, and `tags` with a complex data type object — a collection of key-value pairs with string values. Notice how we have made use of attributes like `description` and `default`.

The `ami` variable also has validation rules defined for them to **check the validity of the value provided**. We have also marked the `type` variable as `sensitive`.

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

 network_interface {
   network_interface_id = aws_network_interface.my_nic.id
   device_index         = 0
 }
}

resource "aws_network_interface" "my_nic" {
 description = "My NIC"
 subnet_id   = var.subnet

 tags = {
   Name = "My NIC"
 }
}
```

Within the *resource* blocks, we have simply used these variables by using `var.<variable name>` format. When you proceed to plan and apply this configuration, the variable values will automatically be replaced by default values. The following is a sample plan output.

```
        }
    }

Plan: 2 to add, 0 to change, 0 to destroy.
```

To check how validation works, modify the default value provided to the `ami` variable. Make sure to change the `ami-` part since validation rules are validating the same. Run

```
|
| Error: Invalid value for variable
|
|   on variables.tf line 1:
|    1: variable "ami" {
|
| Please provide a valid value for variable AMI.
|
| This was checked by the validation rule at variables.tf:6,3-13.
```

Also, notice how the `type` value is represented in the plan output. Since we have marked it as `sensitive`, its value is not shown. Instead, it just displays `sensitive`.

```
+ id                                                   = (known after appl
    + instance_initiated_shutdown_behavior = (known after apply)
    + instance_state                                   = (known after apply)
    + instance_type                                    = (sensitive)
    + ipv6_address_count                             = (known after apply)
    + ipv6_addresses                                   = (known after apply)
```

💡 You might also like:

5 Ways to Manage Terraform at Scale

How to Automate Terraform Deployments and Infrastructure Provisioning

How to Improve Your Infrastructure as Code using Terraform

Variable Substitution using CLI and .tfvars

In the previous example, we relied on the default values of the variables. However, variables are generally used to substitute values during runtime. The default values can be overridden in two ways —

Passing the values in CLI as `-var` argument.
Using .tfvars file to set variable values explicitly.

If we want to initialize the variables using the CLI argument, we can do so as below. Running this command results in Terraform using these values instead of defaults

```
terraform plan -var "ami=test" -var "type=t2.nano" -var "tags={\"name\":\"My Virtua
```

While working with `plan` or `apply` commands, `-var` argument should be used for every variable to be overridden. Note how we have provided the value for complex data type with escaped characters.

Imagine a scenario where there are many variables used in the configuration. Passing the values using CLI arguments can become a tedious task. This is where **.tfvars files come into play**.

Create a file with the `.tfvars` extension and add the below content to it. I have used the name `values.tfvars` as the file name. This way we can organize and manage variable values easily.

```
ami  = "ami-0d26eb3972b7f8c96"
type = "t2.nano"
tags = {
 "name" : "My Virtual Machine"
```

This time, we should ask Terraform to use the `values.tfvars` file by providing its path to `-var-file` CLI argument. The final `plan` command should look as such:

```
terraform plan -var-file values.tfvars
```

The `-var-file` argument is great if you have multiple `.tfvars` files with variations in values. However, if you do not wish to provide the file path every time you run `plan` or `apply`, simply name the file as `<filename>.auto.tfvars`. This file is then automatically chosen to supply input variable values.

Environment Variables

Additionally, input variable values can also be set using Terraform environment variables. To do so, simply set the environment variable in the format `TF_VAR_<variable name>`.

The variable name part of the format is the same as the variables declared in the `variables.tf` file. For example, to set the `ami` variable run the below command to set its corresponding value.

```
export TF_VAR_ami=ami-0d26eb3972b7f8c96
```

Apart from the above environment variable, it is important to note that Terraform also uses a few other environment variables like `TF_LOG`, `TF_CLI_ARGS`, `TF_DATA_DIR`, etc. These environment variables are used for various purposes like logging, setting default behavior with respect to workspaces, CLI arguments, etc.

Precedence

As we have seen till now, there are three ways of providing input values to Terraform configuration using variables. Namely—default values, CLI arguments, and .tfvars file. The **precedence is given to values passed via CLI arguments**. This is followed by values passed using the `.tfvars` file and lastly, the default values are considered.

In the current example, now that we have the `values.tfvars` file saved, try to run a `plan` command by passing values via CLI `-var` arguments. Make sure to provide different values as that of .tfvars and defaults. Terraform ignores the values provided via .tfvars and defaults.

If the values are not provided in the .tfvars file, or as defaults, or as CLI arguments, it falls back on `TF_VAR_` environment variables.

Additionally, if we don't provide the values in any of the forms discussed above, Terraform would ask for the same in interactive mode when `plan` or `apply` commands are run.

As a best practice, it is **not recommended to store secret and sensitive information in variable files**. These values should always be provided via the `TF_VAR_` environment variable.

This is where Spacelift shines. It makes use of these Terraform native environment variables to manage secrets as well as other attributes that make the most sense. Managing the environment in the Spacelift console is easy, thanks to a dedicated tab where values can be edited on the go.

Output Variables

This information is most useful for passing the values to modules along with other scenarios.

This information is also available in Terraform state files. But state files are large, and normally we would have to perform an intricate search for this kind of information.

Output variables in Terraform are used to display the required information in the console output after a successful application of configuration for the root module. To declare an output variable, write the following configuration block into the Terraform configuration files.

```
output "instance_id" {
 value       = aws_instance.myvm.id
 description = "AWS EC2 instance ID"
 sensitive   = false
}
```

Continuing with the same example, we would like to display the instance ID of the EC2 instance that is created. So declare an output variable named `instance_id` — this could be any name of our choosing.

Within this output block, we have used some attributes to associate this output variable's value. We have used resource reference for `aws_instance.myvm` configuration and specified to use its `id` attribute.

Optionally, we can use the `description` and `sensitive` flags. We have discussed the purpose of these attributes in previous sections. When a `plan` command is run, the plan output acknowledges the output variable being declared as below.

```
Changes to Outputs:
```

Similarly, when we run the `apply` command, upon successful creation of EC2 instance, we would know the instance ID of the same. Once the deployment is successful, output variables can also be accessed using the `output` command:

```
terraform output
```

Output:

```
instance_id = "i-xxxxxxxx"
```

Output variables are used by **child modules** to expose certain values to the root module. The root module does not have access to any other component being created by the child module. So, if some information needs to be made available to the root module, output variables should be declared for the corresponding attributes within the child module.

Using Variables in for_each loop

This section goes through an example where we want to **create multiple subnets** for a given VPC. Without variables, we would write the configurations for the number of subnets specified. The other way is to use the `for` loop and **create separate subnets** based on the index value of the iteration.

The *for loop* thus implemented is not very useful. Consider a case where subnets need to be in selected availability zones, and each of them has a different CIDR range specified. For situations like these, we can make use of complex variable type `map(object())` along with `for` loop to iterate over this variable.

we want to define a variable that has a map of objects, with a couple of attributes in each object. We have used `cidr` and `az` attributes for each object with string type.

```
variable "my_subnets" {
 type = map(object({
   cidr = string
   az   = string
 }))
 description = "Subnets for My VPC"
}
```

Let us move to the .tfvars file to initialize the value for this input variable. The `.tfvars` file should contain the following lines of code.

```
my_subnets = {
 "a" = {
   cidr = "10.0.1.0/26"
   az   = "eu-central-1a"
 },
 "b" = {
   cidr = "10.0.2.0/26"
   az   = "eu-central-1a"
 },
 "c" = {
   cidr = "10.0.3.0/26"
   az   = "eu-central-1b"
 },
 "d" = {
   cidr = "10.0.4.0/26"
   az   = "eu-central-1c"
 },
 "e" = {
   cidr = "10.0.5.0/26"
   az   = "eu-central-1b"
```

If you compare the input variable declaration and initialization, you will see we have aligned the attributes. Also, the initialized value consists of 6 objects mapped by a string key. Each object has a unique CIDR and Availability Zone (`az`) value.

Lastly, let's define the configuration for the subnets themselves. Note: The following code assumes that we have defined `aws_vpc.my_vpc` elsewhere in the configuration.

```
resource "aws_subnet" "my_subnets" {
 for_each          = var.my_subnets
 vpc_id            = aws_vpc.my_vpc.id
 cidr_block        = each.value.cidr
 availability_zone = each.value.az

 tags = {
   Name = "Subnet - ${each.value.az}"
 }
}
```

In this resource block, we have used the `my_subnets` variable to iterate over in a `for_each` loop. `for_each` loop comes along with a keyword `each` , which helps us identify the value to be assigned in each iteration. This single block of code is capable of creating 6 unique subnets.

Limitations

It is important to note that Terraform does not allow the usage of variables in provider configuration blocks. This is mainly to adhere to best practices of using Terraform.

Variables usually make the developer's life easier by **improving the maintainability of your code**. Especially in larger Terraform codebases, variables should be used. Larger

Working in a team of Terraform developers can be challenging, especially for state files. Spacelift is built to **provide a great CI/CD experience concerning IaC** where it is used for version control of the code as well as state management. It is a perfect place to set up a CI/CD pipeline for Terraform code. If you are struggling with Terraform automation and management, check out Spacelift. You can sign up for a free evaluation right away.

I hope this blog post was helpful to you in understanding and exploring possibilities with Terraform variables. Let me know your thoughts in the comments section.

Written by

Sumeet Ninawe

specializes in writing IaC using Terraform. In his free time, Samuel maintains a blog at [illegible].

spacelift

Product

Documentation

How it works

Spacelift Tutorial

Pricing

Customer Case Studies

Integrations

Security

System Status

Product Updates

Company

About Us

Careers

Contact Sales

Partners

Learn

Blog

Spacelift vs Atlantis

Spacelift vs Terraform Cloud

Get our newsletter

Subscribe





Privacy Policy Terms of Service