# Terraform variable validation

Jack Roper Follow
Sep 16 · 5 min read ★

In this post, I'll dive into variable validation in Terraform, why you should do it, and show lots of useful examples!


## Why validate your variables?

Although the syntax and configuration of your Terraform may be valid, the variables passed into your configuration may not be valid. Passing invalid variables generally results in errors during the deployment stage.

Examples of this might be setting an invalid VM size, an invalid IP address, or checking your environment variable matches one of 'dev', 'uat', or 'prod'.

In addition, without validation, the error message displayed to the user when an invalid variable is passed in comes directly from the Azure API, which can sometimes be hard to read and might not spell out what exactly was wrong with the variable. With validation in Terraform, you can specify the error message.

Note that the validation error message must be at least one full English sentence starting with an uppercase letter and ending with a period or question mark.

Variable validation is available in Terraform v0.13 and above.

### Validation syntax

Typically, a variable in Terraform is defined in the example below for an API token. It defines the type as a string and adds a description.

```
variable "some_api_token" {
  type        = string
  description = "API token"
}
```

A user could define the variable as `api_token = ""` (an empty string) or `api_token = "badtoken123"` (an invalid token) which would both result in an error during the deployment. If the API Token should always be 32 characters long, we can add a validation block with a condition to check this, and define the error message if this condition is not met:

```
variable "some_api_token" {
  type        = string
  description = "API token"
  validation {
    condition     = length(var.some_api_token) == 32
    error_message = "Must be a 32 character long API token."
  }
}
```

To test this, you can try applying the configuration and passing in an invalid token and you should see the error message:

```
terraform apply -target=null_resource.validate_some_api_token -var
"some_api_token=xxx"
```

If the expression defined in the validation block returns `true` then the validation is considered to have succeeded. If it returns `false` then validation has failed and Terraform will return an error at the module call site, using the text given in `error_message.`

### Validate IP address variables

For more complex validations, you can use Regex. You can test regex expressions here.

A really useful and common validation condition can be used to check the IP addresses you pass in are valid:

```
variable "ip_address" {
    type          = string
    description   = "Example to validate IP address."
    validation {
        condition = can(regex("^(25[0-5]|2[0-4][0-9]|[01]?[0-9][0-
9]?)\\.(25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?)\\.(25[0-5]|2[0-4][0-9]|
[01]?[0-9][0-9]?)\\.(25[0-5]|2[0-4][0-9]|[01]?[0-9][0-
9]?)$",var.ip_address))
        error_message = "Invalid IP address provided."
    }
}
```

Here we are using the `can` function which evaluates the regex expression and returns a boolean value indicating whether the expression produced a result without any errors.

This would flag any invalid inputs such as :

- “192.0.1.” (missing the last octet)
- “192.65.0.**256**” (outside of 1–255)
- “10.0.10,180” (comma instead of .)

The following would validate a list of IP Addresses:

```
variable "ip_address_list" {
    type          = string
    description   = "Example to validate list of IP addresses."
    validation {
        condition = can([for ip in var.ip_address_list:
regex("^(25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?)\\.(25[0-5]|2[0-4][0-
9]|[01]?[0-9][0-9]?)\\.(25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?)\\.
(25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?)$", ip)])
        error_message = "Invalid List of IP addresses provided."
    }
}
```

Invalid inputs would include: [“192.65.0.256”, “192.65.017”, “192.50,0.18”]

### Validate a timestamp

```
variable "timestamp" {
  type        = string
  description = "Example to validate a timestamp"

  validation {
    condition     = can(formatdate("", var.timestamp))
    error_message = "The timestamp argument requires a valid RFC 3339
timestamp."
  }
}
```

Here `formatdate` is used in conjunction with `can` which will fail if the second argument is not a valid timestamp.

### Validate from a list of allowed values, e.g. VM Size

Adding a condition with a list of allowed values is useful in many scenarios. For example, checking the VM size is as desired (You could also use Azure Policy to enforce this, validation at this level may not always be ideal). In the 3 examples below we check that the VM size is set to one of 3 values, “Standard_DS2”, “Standard_D2” or “Standard_DS2_v2” :

```
variable "vm_size" {
  type        = string
  description = "VM Size"

  validation {
    condition = anytrue([
      var.env == "Standard_DS2",
      var.env == "Standard_D2",
      var.env == "Standard_DS2_v2"
    ])
    error_message = "VM Size must be "Standard_DS2", "Standard_D2" or
"Standard_DS2_v2".
  }
}
```

Another way to achieve this without the `anytrue` function:

```
variable "vm_size" {
  type        = string
  description = "VM Size"

  validation {
    condition     = contains(["Standard_DS2", "Standard_D2",
"Standard_DS2_v2"], var.vm_size)
    error_message = "VM Size must be "Standard_DS2", "Standard_D2" or
"Standard_DS2_v2".
  }
}
```

Another way with the OR operator:

```
variable "vm_size" {
  type        = string
  description = "VM Size"

validation {
    condition     = var.vm_size == "Standard_DS2" ||var.vm_size ==
"Standard_D2"||var.vm_size == "Standard_DS2_v2"
    error_message = "VM Size must be "Standard_DS2", "Standard_D2" or
"Standard_DS2_v2".
  }
}
```

### Validate an Azure Storage account name

Azure storage account names must be globally unique and between 3 to 24 characters. Although we can't check it will be globally unique, we can validate the account name is between 3 and 24 characters:

```
variable "storage_account_name" {
  type        = string
  description = "Globally unique name for the storage account."

    validation {
      condition     = length(var.storage_account_name) >=3 &&
length(var.storage_account_name) <= 24
      error_message = "The storage_account_name variable name must be
3-24 characters in length."
    }
}
```

### Validate an AMI ID for a VM (on AWS)

```
variable "ami_id" {
  type = string

  validation {
    condition = (
      length(var.ami_id) > 4 &&
      substr(var.ami_id, 0, 4) == "ami-"
    )
    error_message = "The ami_id value must start with \"ami-\"."
  }
}
```

This requires the string to start with “ami-”.

## Closing

Using the Terraform variable validation function can help you avoid errors at deployment time. You can make use of and combine the various built-in Terraform functions to achieve the result you want and display a meaningful error message.

To learn more check out the Terraform docs:

> Input Variables - Configuration Language - Terraform by HashiCorp
>
> Hands-on: Try the Customize Terraform Configuration with Variables tutorial on HashiCorp Learn. Input variables serve...
>
> www.terraform.io


Cheers!



Azure | Terraform | Azure Devops | Infrastructure As Code | AWS