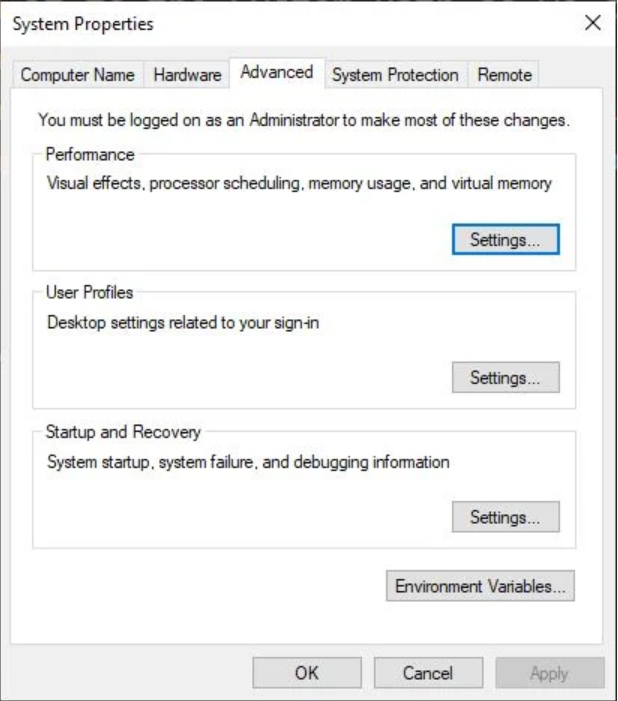
**TERRAFORM**

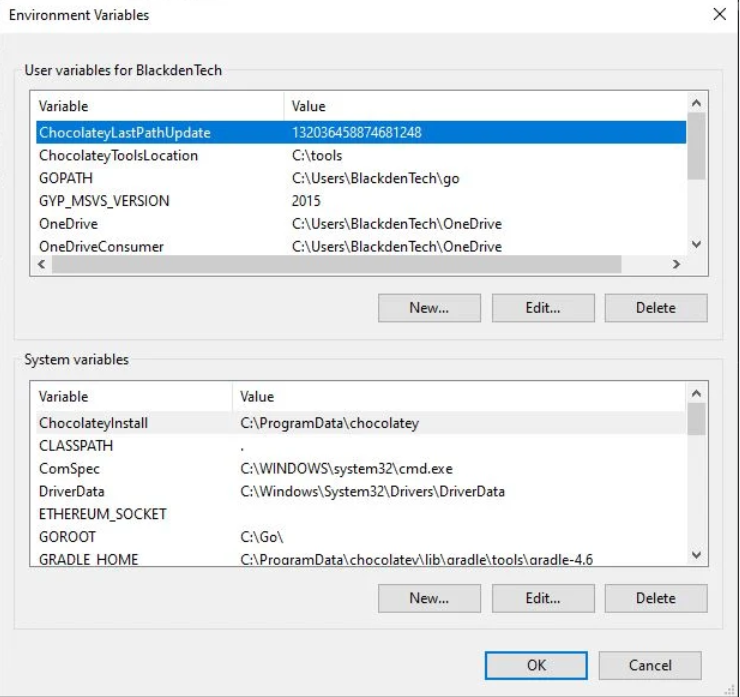
Terraform is an open-source infrastructure as code software tool created by HashiCorp. Users define and provide data-centre infrastructure using a declarative configuration language known as HashiCorp Configuration Language (hcl).

**Installation of Terraform: -**

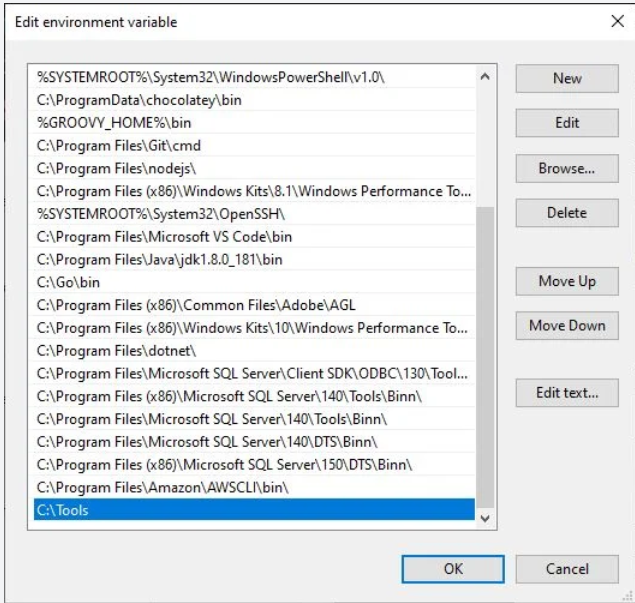
1. Download the appropriate version of Terraform from [HashiCorp’s download page](https://www.terraform.io/downloads.html). In my case, it’s the Windows 64-bit version.
2. Make a folder on your C:\ drive where you can put the Terraform executable.
3. After the download finishes, go find it in File Explorer. Extract the zip file to the folder you created in step 2.
4. Open your Start Menu and type in “environment” and the first thing that comes up should be Edit the System Environment Variables option. Click on that and you should see this window:



1. Click on Environment Variables… at the bottom and you’ll see this:



1. Under the bottom section where it says System Variables, find one called Path and click edit. You’ll then see a list of where to find the binaries that Windows might need for any given reason.



1. Click OK on each of the menus you’ve opened up until there’s no more left.
2. Verify the installation was successful by entering terraform --version. If it returns a version, you’re good to go.

**For linux users:**

1. wget <https://releases.hashicorp.com/terraform/1.0.4/terraform_1.0.4_linux_amd64.zip>
2. unzip terraform-zipped file
3. /terraform: export PATH=$PATH:/home/ec2-user/terraform
4. Terraform version

**COMMANDS**

* init:

1. Used to initialize a working directory containing terraform config file.
2. Downloads provider

* validate

Validates terraform config files and ensure that they are syntactically valid and internally consistent.

* plan

creates execution plan

* apply

apply checks the current directory for the configuration and applies the changes appropriately

* destroy

used to destroy the terraform managed infrastructure

To destroy a particular resource: terraform destroy -target aws\_instance.web

**CONFIGURATION SYNTAX FOR TERRAFORM**

# Template

<BLOCK TYPE> "<BLOCK LABEL>" "<BLOCK LABEL >" {

# Block body

#Argument Name #Argument Value

<IDENTIFIER> = <EXPRESSION> # Argument

}

# AWS Example

resource "aws\_instance" "example" { # BLOCK

ami = "ami-04d29b6f966df1537" # Argument

instance\_type = var.instance\_type # Argument with value as expression (Variable value replaced from varibales.tf)

network\_interface {

# ...

}

}

**BLOCKS**: A block has a type (resource in this example). Each block type defines how many labels must follow the type keyword. The resource block type expects two labels, which are aws\_instance and example in the example above. A particular block type may have any number of required labels, or it may require none as with the nested network\_interface block type.

After the block type keyword and any labels, the block body is delimited by the { and } characters. Within the block body, further arguments and blocks may be nested, creating a hierarchy of blocks and their associated arguments.

The Terraform language uses a limited number of top-level block types, which are blocks that can appear outside of any other block in a configuration file. Most of Terraform's features (including resources, input variables, output values, data sources, etc.) are implemented as top-level blocks.

There are two types of BLOCKS:

1. Top level block
2. Block inside block (Nested block)

**Example for top level blocks is**: resource, provider, variable etc.

**Example for block inside block is**: provisioner, resource specific blocks like tags, network\_interface

**RESOURCE NAMES**: Resource names are nouns, since resource blocks each represent a single object Terraform is managing. Resource names must always start with their containing provider's name followed by an underscore, so a resource from the provider postgresql might be named postgresql\_database.

**Arguments:** Arguments configure a particular resource; because of this, many arguments are resource-specific. Arguments can be required or optional, as specified by the provider. If you do not supply a required argument, terraform will give an error and not apply the configuration.

**Attributes**: Attributes are values exposed by an existing resource. References to resource attributes take the format **resource\_type.resource\_name.attribute\_name**. Unlike arguments which specify an infrastructure object's configuration, a resource's attributes are often assigned to it by the underlying cloud provider or API.

**Identifier**: Argument names, block type names, and the names of most Terraform-specific constructs like resources, input variables, etc. are all identifiers. Identifiers can contain letters, digits, underscores (\_), and hyphens (-). The first character of an identifier must not be a digit, to avoid ambiguity with literal numbers.

**Meta-arguments** change a resource's behaviour, such as using a count meta-argument to create multiple resources. Meta-arguments are a function of Terraform itself and are not resource or provider-specific.

Meta Arguments are: -

[depends\_on](https://www.terraform.io/docs/language/meta-arguments/depends_on.html)

[count](https://www.terraform.io/docs/language/meta-arguments/count.html)

[for\_each](https://www.terraform.io/docs/language/meta-arguments/for_each.html)

[provider](https://www.terraform.io/docs/language/meta-arguments/resource-provider.html)

[lifecycle](https://www.terraform.io/docs/language/meta-arguments/lifecycle.html)

**Links**: -

[Additional Reference](https://learn.hashicorp.com/tutorials/terraform/resource?in=terraform/configuration-language)

[Resource: AWS Instance](https://registry.terraform.io/providers/hashicorp/aws/latest/docs/resources/instance)

[Resource: AWS Instance Argument Reference](https://registry.terraform.io/providers/hashicorp/aws/latest/docs/resources/instance#argument-reference)

[Resource: AWS Instance Attribute Reference](https://registry.terraform.io/providers/hashicorp/aws/latest/docs/resources/instance#attributes-reference)

[Resource: Meta-Arguments](https://www.terraform.io/docs/language/meta-arguments/depends_on.html)

**COMMENTS IN TERRAFORM**

* Single line comment:

Single line comment can be done using: # or //

* Multi line comment:

Multi line comment can be done using:

/\*

Line -1

Line-2

\*/

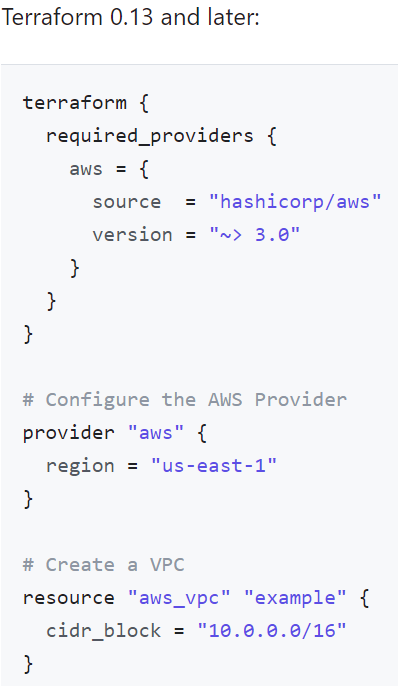
**TERRAFORM TOP LEVEL BLOCK**

Terraform language uses a limited number of top-level block types, which are blocks that can appear outside of any other block in a TF configuration file. Most of Terraform’s features are implemented as top-level blocks.

* Fundamental Blocks: Terraform Block, Providers Block, Resources Block
* Variable Blocks: Input variables, Output Variables, Local Values Block
* Calling/Referencing Blocks: Data Sources Block, Modules Block

**Terraform Block: -**

* This particular block used to configure Terraform Version, required providers and Terraform backend.
* Each Terraform block can contain a number of settings related to Terraform behaviour.
* Within a Terraform block, only constant values can be used, arguments may not refer to named objects such as resources, input variables etc. and may not use any of the terraform built-in functions.

# Block-1: Terraform Settings Block

terraform {

required\_version = "~> 0.14"

required\_providers {

aws = {

source = "hashicorp/aws"

version = "~> 3.0"

}

}

# Adding Backend as S3 for Remote State Storage with State Locking

backend "s3" {

bucket = "terraform-stacksimplify"

key = "dev2/terraform.tfstate"

region = "us-east-1"

# For State Locking

dynamodb\_table = "terraform-dev-state-table"

}

}

**Providers Block: -**

* Terraform relies on providers to interact with Remote Systems.
* Declare providers for terraform to install providers & use them.

# Block-2: Provider Block

provider "aws" {

profile = "default"

region = "us-east-1"

}

**Components of providers’ block: -**

* Provider Requirements:

**required\_providers {**

**aws = {**

**source = "hashicorp/aws"**

**version = "3.44.0"**

**}**

**}**

* Provider Configuration:

**provider "aws" {**

**region = "ap-south-1"**

**shared\_credentials\_file = "/c/Users/DELL/.aws/creds"**

**profile = "developer"**

**}**

* Dependency Lock File

**.terraform.lock.hcl**

**PROVIDER VERSIONING**

During terraform init, if version argument is not specified, the most recent provider will be downloaded during initialization.

For Production use, you should constrain the acceptable provider via configuration, to ensure that new versions with breaking changes will not be automatically installed.

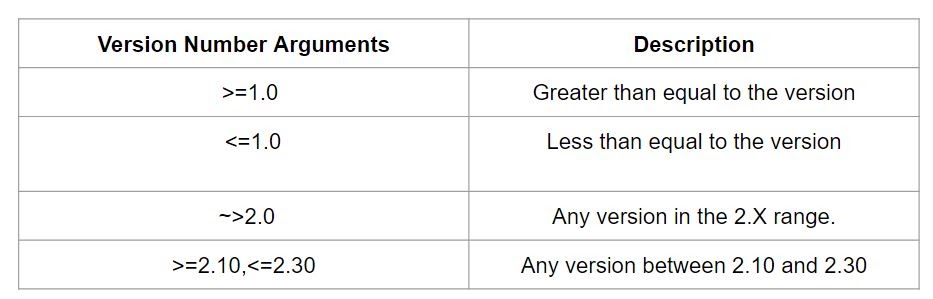
**Dependency Lock File:**

Terraform dependency lock file allows us to lock to a specific version of the provider.

If a particular provider already has a selection recorded in the lock file, terraform will always re-select that version for installation, even if a newer version has become available.

You can override that behaviour by adding the -upgrade option when you run terraform init.

**There are multiple ways of specifying the version of a provider**:



**Resource Block: -**

Resourcesare the most important element in the Terraform language. Eachresource block describes one or more infrastructure objects, such as virtual networks, compute instances, or higher-level components such as DNS records.a

#Block-3: Resource Block

resource "aws\_instance" "ec2demo" {

ami = "ami-04d29b6f966df1537" # Amazon Linux

instance\_type = var.instance\_type

}

**Resource Type**: It determines the kind of infrastructure object it manages and what arguments and other attributes the resource supports.

**Resource Local Name**: It is used to refer to this resource from elsewhere in the same terraform module, but has no significance outside that module’s scope.

The resource type and name together server as an identifier for a given resource and so must be unique within a module.

**USER AUTHENTICATION**

**1st way:**

With default profile credentials is not required to be configured.

Default credentials can be used: -

aws configure

**2nd way:**

Hardcoding credentials into config file:

provider "aws" {

region = "us-west-2"

access\_key = "my-access-key"

secret\_key = "my-secret-key"

}

**3rd way:**

Using as environment variables

$ export AWS\_ACCESS\_KEY\_ID="anaccesskey"

$ export AWS\_SECRET\_ACCESS\_KEY="asecretkey"

**4th way:**

Providing credentials file location: -

provider "aws" {

region = "us-west-2"

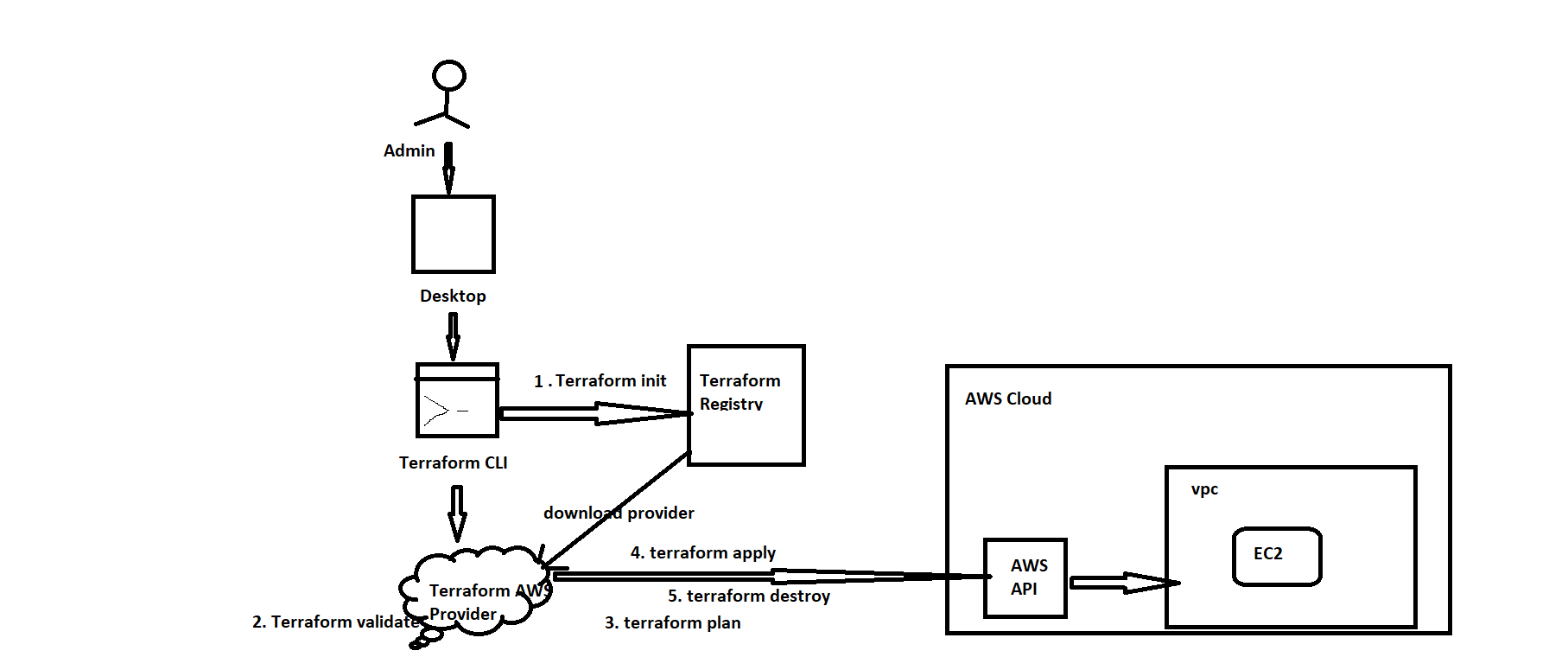
shared\_credentials\_file = "/Users/tf\_user/.aws/creds"

profile = "customprofile"

}

**5th way:**

Creating IAM role for AWS instance on which terraform is running

****

**Terraform state**

Terraform stores the state of the infrastructure that is being created from the TF file.

This state allows terraform to map real world resource to your existing configuration.

Terraform state file get created for first time when we run terraform apply command. It maintains real infrastructure information. It is underlying database for current project.

**Desired State VS Current State in Terraform**

**Desired State**

Terraform’s primary function is to create, modify and destroy infrastructure resources to match to desired state described in Terraform configuration

**Current State**

Current state is actual state of a resource that is currently deployed

**Note:** Terraform tries to ensure that the deployed infrastructure is based on the desired state.

If there is a difference between the two, terraform plan presents a description of the changes necessary to achieve the desired state.

Project-1:

**VARIABLES**

**Terraform Input Variables: -**

Input variables serve as parameters for a Terraform module, allowing aspects of the module to be customized without altering the module's own source code, and allowing modules to be shared between different configurations.

It contains:

* [default](https://www.terraform.io/docs/language/values/variables.html#default-values) - A default value which then makes the variable optional.
* [type](https://www.terraform.io/docs/language/values/variables.html#type-constraints) - This argument specifies what value types are accepted for the variable.
* [description](https://www.terraform.io/docs/language/values/variables.html#input-variable-documentation) - This specifies the input variable's documentation.
* [validation](https://www.terraform.io/docs/language/values/variables.html#custom-validation-rules) - A block to define validation rules, usually in addition to type constraints.
* [sensitive](https://www.terraform.io/docs/language/values/variables.html#suppressing-values-in-cli-output) - Limits Terraform UI output when the variable is used in configuration.

Syntax:

1. By dynamic input

variable "instance\_type" {

  type = string

  description = "enter the instance type"

}

1. You can also assign default value to variables and if explicitly you are not providing value, then default value will be used. For ex:

variable "instance\_type" {

  default = "t2.micro"

}

terraform plan -var=”instance\_type=t2.nano”

1. From a file:

You can store value in a file terraform.tfvars file

instance\_type="t2.medium"

if you wish to use a customized file then while running command provide the file name

terraform plan -var-file=”abc.tfvars”

1. Using environment variables:

export TF\_VAR\_<Variable-name>=”<value>”

**Type in variable:**

* String: Sequence of Unicode characters representing some text, “hello”
* List: [“t2.micro”,”t2.large”,”t3.nano”]
* Map {instance\_type=”t2.micro”,owner= 799565682383}
* Number 100

**Fetching data from map and list:-**

resource "aws\_instance" "web" {

  ami = "ami-0e59237271d3dd068"

  #instance\_type = var.instance\_type\_list[1]

  instance\_type = var.instance\_type\_map["dev"]

}

/\*

variable "instance\_type\_list" {

  type = list

  default = ["t2.micro","t2.nano","t2.large"]

}

\*/

variable "instance\_type\_map" {

  type = map

  default = {

    dev="t2.nano"

    qa="t2.micro"

    prod="t3.large"

  }

}

**Terraform Data sources: -**

data sources are read-only views into the state of pre-existing components external to our configuration. Once you have defined a data source, you can use the data elsewhere in your Terraform configuration.

Syntax**:**

data "aws\_ami" "example" {

  most\_recent      = true

  owners           = ["amazon"]

  filter {

    name   = "root-device-type"

    values = ["ebs"]

  }

  filter {

    name   = "virtualization-type"

    values = ["hvm"]

  }

}

**Terraform Output Values: -**

Terraform output values allow you to export structured data about your resources. You can use this data to configure other parts of your infrastructure with automation tools, or as a data source for another Terraform workspace. Outputs are also necessary to share data from a child module to your root module.

**Syntax**:

output "private\_ip" {

  value=aws\_instance.web.private\_ip

  sensitive = false

}

Commands to check output values:

**terraform output**

**terraform output <output name>**

**terraform output -json**

**Meta arguments**

* 1. Count: count is a meta-argument defined by the Terraform language. It can be used with modules and with every resource type. The count meta-argument accepts a whole number, and creates that many instances of the resource or module. Each instance has a distinct infrastructure object associated with it, and each is separately created, updated, or destroyed when the configuration is applied.

Example:

* provider "aws" {
* region = "ap-south-1"
* profile = "developer"
* }
* resource "aws\_instance" "web" {
* ami = "ami-0e59237271d3dd068"
* #instance\_type = var.instance\_type\_list[1]
* instance\_type = var.instance\_type\_map["dev"]
* count=4
* }
* /\*
* variable "instance\_type\_list" {
* type = list
* default = ["t2.micro","t2.nano","t2.large"]
* }
* \*/
* variable "instance\_type\_map" {
* type = map
* default = {
* dev="t2.nano"
* qa="t2.micro"
* prod="t3.large"
* }
* }

1. For & Splat: -

The for loop in terraform is similar to for loop in any other language, with for loop you can iterate over a collection (list, Map).

**Note**: We can use a similar approach to iterate over the map also. But always keep in mind you need to specify the type of the map like string or number.

{for <KEY>, <VALUE> in <MAP> : <OUTPUT\_KEY> => <OUTPUT\_VALUE>}

Syntax:

[for <ITEM> in <LIST> : <OUTPUT\_VALUE> ]

{for <KEY>, <VALUE> in <MAP> : <OUTPUT\_KEY> => <OUTPUT\_VALUE>}

Splat Expressions:

A splat expression provides a more concise way to express a common operation that could otherwise be performed with a for expression.

If var.list is a list of objects that all have an attribute id, then a list of the ids could be produced with the following for expression:

[for o in var.list : o.id]

This is equivalent to the following splat expression:

var.list[\*].id

The special [\*] symbol iterates over all of the elements of the list given to its left and accesses from each one the attribute name given on its right.

A splat expression can also be used to access attributes and indexes from lists of complex types by extending the sequence of operations to the right of the symbol:

var.list[\*].interfaces[0].name

The above expression is equivalent to the following for expression:

[for o in var.list : o.interfaces[0].name]

»Splat Expressions with Maps

The splat expression patterns shown above apply only to lists, sets, and tuples. To get a similar result with a map or object value you must use for expressions.

Resources that use the for\_each argument will appear in expressions as a map of objects, so you can't use splat expressions with those resources. For more information, see Referring to Resource Instances.

»Single Values as Lists

Splat expressions have a special behaviour when you apply them to a value that isn't a list, set, or tuple.

If the value is anything other than a null value then the splat expression will transform it into a single-element list, or more accurately a single-element tuple value. If the value is null then the splat expression will return an empty tuple.

This special behaviour can be useful for modules that accept optional input variables whose default value is null to represent the absense of any value, to adapt the variable value to work with other Terraform language features that are designed to work with collections. For example:

variable "website" {

type = object({

index\_document = string

error\_document = string

})

default = null

}

resource "aws\_s3\_bucket" "example" {

# ...

dynamic "website" {

for\_each = var.website[\*]

content {

index\_document = website.value.index\_document

error\_document = website.value.error\_document

}

}

}

The above example uses a dynamic block, which generates zero or more nested blocks based on a collection value. The input variable var.website is defined as a single object that might be null, so the dynamic block's for\_each expression uses [\*] to ensure that there will be one block if the module caller sets the website argument, or zero blocks if the caller leaves it set to null.

This special behaviour of splat expressions is not obvious to an unfamiliar reader, so we recommend using it only in for\_each arguments and similar situations where the context implies working with a collection. Otherwise, the meaning of the expression may be unclear to future readers.

»Legacy (Attribute-only) Splat Expressions

Earlier versions of the Terraform language had a slightly different version of splat expressions, which Terraform continues to support for backward compatibility. This older variant is less useful than the modern form described above, and so we recommend against using it in new configurations.

The legacy "attribute-only" splat expressions use the sequence .\*, instead of [\*]:

var.list.\*.interfaces[0].name

This form has a subtly different behavior, equivalent to the following for expression:

[for o in var.list : o.interfaces][0].name

Notice that with the attribute-only splat expression the index operation [0] is applied to the result of the iteration, rather than as part of the iteration itself. Only the attribute lookups apply to each element of the input. This limitation was confusing some people using older versions of Terraform and so we recommend always using the new-style splat expressions, with [\*], to get the more consistent behavior.

1. For-each

for\_each is a meta-argument defined by the Terraform language. It can be used with modules and with every resource type.

The for\_each meta-argument accepts a map or a set of strings, and creates an instance for each item in that map or set. Each instance has a distinct infrastructure object associated with it, and each is separately created, updated, or destroyed when the configuration is applied.

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 Before for\_each was available, it was common to derive count from the length of a list and use count.index to look up the original list value:

variable "subnet\_ids" {

type = list(string)

}

resource "aws\_instance" "server" {

# Create one instance for each subnet

count = length(var.subnet\_ids)

ami = "ami-a1b2c3d4"

instance\_type = "t2.micro"

subnet\_id = var.subnet\_ids[count.index]

tags = {

Name = "Server ${count.index}"

}

}

This was fragile, because the resource instances were still identified by their index instead of the string values in the list. If an element was removed from the middle of the list, every instance after that element would see its subnet\_id value change, resulting in more remote object changes than intended. Using for\_each gives the same flexibility without the extra churn.

**toset Function**

toset converts its argument to a set value.

Explicit type conversions are rarely necessary in Terraform because it will convert types automatically where required. Use the explicit type conversion functions only to normalize types returned in module outputs.

Pass a list value to toset to convert it to a set, which will remove any duplicate elements and discard the ordering of the elements.

[»](https://www.terraform.io/docs/language/functions/toset.html" \l "examples)Examples

> toset(["a", "b", "c"])

[

"a",

"b",

"c",

]

Since Terraform's concept of a set requires all of the elements to be of the same type, mixed-typed elements will be converted to the most general type:

> toset(["a", "b", 3])

[

"a",

"b",

"3",

]

Set collections are unordered and cannot contain duplicate values, so the ordering of the argument elements is lost and any duplicate values are coalesced:

> toset(["c", "b", "b"])

[

"b",

"c",

]

**tomap Function**

tomap converts its argument to a map value.

Explicit type conversions are rarely necessary in Terraform because it will convert types automatically where required. Use the explicit type conversion functions only to normalize types returned in module outputs.

[»](https://www.terraform.io/docs/language/functions/tomap.html#examples)Examples

> tomap({"a" = 1, "b" = 2})

{

"a" = 1

"b" = 2

}

Since Terraform's concept of a map requires all of the elements to be of the same type, mixed-typed elements will be converted to the most general type:

> tomap({"a" = "foo", "b" = true})

{

"a" = "foo"

"b" = "true"

}

# **Splat Expressions**

A splat expression provides a more concise way to express a common operation that could otherwise be performed with a for expression.

If var.list is a list of objects that all have an attribute id, then a list of the ids could be produced with the following for expression:

[for o in var.list : o.id]

This is equivalent to the following splat expression:

var.list[\*].id

The special [\*] symbol iterates over all of the elements of the list given to its left and accesses from each one the attribute name given on its right. A splat expression can also be used to access attributes and indexes from lists of complex types by extending the sequence of operations to the right of the symbol:

var.list[\*].interfaces[0].name

The above expression is equivalent to the following for expression:

[for o in var.list : o.interfaces[0].name]

**Splat Expressions with Maps**

The splat expression patterns shown above apply only to lists, sets, and tuples. To get a similar result with a map or object value you must use [for expressions](https://www.terraform.io/docs/language/expressions/for.html).

Resources that use the for\_each argument will appear in expressions as a map of objects, so you can't use splat expressions with those resources. For more information, see [Referring to Resource Instances](https://www.terraform.io/docs/language/meta-arguments/for_each.html#referring-to-instances).

Single Values as Lists

Splat expressions have a special behavior when you apply them to a value that isn't a list, set, or tuple.

If the value is anything other than a null value then the splat expression will transform it into a single-element list, or more accurately a single-element tuple value. If the value is null then the splat expression will return an empty tuple.

This special behavior can be useful for modules that accept optional input variables whose default value is null to represent the absense of any value, to adapt the variable value to work with other Terraform language features that are designed to work with collections. For example:

variable "website" {

type = object({

index\_document = string

error\_document = string

})

default = null

}

resource "aws\_s3\_bucket" "example" {

# ...

dynamic "website" {

for\_each = var.website[\*]

content {

index\_document = website.value.index\_document

error\_document = website.value.error\_document

}

}

}

The above example uses a [dynamic block](https://www.terraform.io/docs/language/expressions/dynamic-blocks.html), which generates zero or more nested blocks based on a collection value. The input variable var.website is defined as a single object that might be null, so the dynamic block's for\_each expression uses [\*] to ensure that there will be one block if the module caller sets the website argument, or zero blocks if the caller leaves it set to null.

This special behavior of splat expressions is not obvious to an unfamiliar reader, so we recommend using it only in for\_each arguments and similar situations where the context implies working with a collection. Otherwise, the meaning of the expression may be unclear to future readers.

[»](https://www.terraform.io/docs/language/expressions/splat.html" \l "legacy-attribute-only-splat-expressions)Legacy (Attribute-only) Splat Expressions

Earlier versions of the Terraform language had a slightly different version of splat expressions, which Terraform continues to support for backward compatibility. This older variant is less useful than the modern form described above, and so we recommend against using it in new configurations.

The legacy "attribute-only" splat expressions use the sequence .\*, instead of [\*]:

var.list.\*.interfaces[0].name

This form has a subtly different behavior, equivalent to the following for expression:

[for o in var.list : o.interfaces][0].name

Notice that with the attribute-only splat expression the index operation [0] is applied to the result of the iteration, rather than as part of the iteration itself. Only the attribute lookups apply to each element of the input. This limitation was confusing some people using older versions of Terraform and so we recommend always using the new-style splat expressions, with [\*], to get the more consistent behavior.

**MODULES**

A Terraform module is **a set of Terraform configuration files in a single directory**. Even a simple configuration consisting of a single directory with one or more .tf files is a module. When you run Terraform commands directly from such a directory, it is considered the root module.

Modules are the main way to package and reuse resource configurations with Terraform.

In addition to modules from the local file system, Terraform can load modules from a public or private registry.