TerraForm

Terraform is a tool that allows you to define infrastructure in human and machine-readable code

**IAC- Infrastructure as a code**

IaC makes it easy to provision and apply infrastructure configurations, saving time. It standardizes workflows across different infrastructure providers (e.g., VMware, AWS, Azure, GCP, etc.) by using a common syntax across all of them.

**IAC makes changes idempotent, consistent, repeatable, and predictable. Without IAC, scaling up infrastructure to meet increased demand may require an operator to remotely connect to each machine and then manually provision and configure many servers by executing a series of commands/scripts.**

The idempotent characteristic provided by IaC tools ensures that, even if the same code is applied multiple times, the result remains the same.

**IAC Makes Infrastructure More Manageable**

Leveraging HashiCorp Terraform IaC provides benefits that enable mutation, when necessary, via code. Consider an environment has been provisioned that contains a couple servers and a load balancer. To address increased load, additional servers are needed. The IaC can be revised, with minimal changes, to bring new servers online using the previously defined configuration.

During execution, terraform will examine the state of the currently running infrastructure, determine what differences exist between the current state and the revised desired state, and indicate the necessary changes that must be applied. When approved to proceed, only the necessary changes will be applied, leaving existing, valid infrastructure untouched.

## Key Features

### [»](https://www.terraform.io/intro/index.html" \l "infrastructure-as-code)Infrastructure as Code

You describe your infrastructure using Terraform's high-level [configuration language](https://www.terraform.io/docs/language/index.html) in human-readable, declarative configuration files. This allows you to create a blueprint that you can version, share, and reuse.

### [»](https://www.terraform.io/intro/index.html" \l "execution-plans)Execution Plans

Terraform generates an execution plan describing what it will do and asks for your approval before making any infrastructure changes. This allows you to review changes before Terraform creates, updates, or destroys infrastructure.

### [»](https://www.terraform.io/intro/index.html" \l "resource-graph)Resource Graph

Terraform builds a resource graph and creates or modifies non-dependent resources in parallel. This allows Terraform to build resources as efficiently as possible and gives you greater insight into your infrastructure.

### [»](https://www.terraform.io/intro/index.html" \l "change-automation)Change Automation

Terraform can apply complex changesets to your infrastructure with minimal human interaction. When you update configuration files, Terraform determines what changed and creates incremental execution plans that respect dependencies.

# Use Cases

This page lists a subset of use cases for [Terraform](https://www.terraform.io/intro/index.html).

**[»](https://www.terraform.io/intro/use-cases.html" \l "heroku-app-setup)Heroku App Setup**

Heroku is a popular PaaS for hosting web apps. Developers create an app, and then attach add-ons, such as a database, or email provider. One of the best features is the ability to elastically scale the number of dynos or workers.

**[»](https://www.terraform.io/intro/use-cases.html" \l "multi-tier-applications)Multi-Tier Applications**

A very common pattern is the N-tier architecture. The most common 2-tier architecture is a pool of web servers that use a database tier. Additional tiers get added for API servers, caching servers, routing meshes, etc. This pattern is used because the tiers can be scaled independently and provide a separation of concerns.

**[»](https://www.terraform.io/intro/use-cases.html" \l "self-service-clusters)Self-Service Clusters**

At a certain organizational size, it becomes very challenging for a centralized operations team to manage a large and growing infrastructure. Instead it becomes more attractive to make "self-serve" infrastructure, allowing product teams to manage their own infrastructure using tooling provided by the central operations team.

**[»](https://www.terraform.io/intro/use-cases.html" \l "software-demos)Software Demos**

Software writers can provide a Terraform configuration to create, provision and bootstrap a demo on cloud providers like AWS. This allows end users to easily demo the software on their own infrastructure, and even enables tweaking parameters like cluster size to more rigorously test tools at any scale.

**[»](https://www.terraform.io/intro/use-cases.html" \l "disposable-environments)Disposable Environments**

It is common practice to have both a production and staging or QA environment. Using Terraform, the production environment can be codified and then shared with staging, QA or dev. These configurations can be used to rapidly spin up new environments to test in, and then be easily disposed of.

**[»](https://www.terraform.io/intro/use-cases.html" \l "software-defined-networking)Software Defined Networking**

Software Defined Networking (SDN) is becoming increasingly prevalent in the datacenter, as it provides more control to operators and developers and allows the network to better support the applications running on top. Most SDN implementations have a control layer and infrastructure layer.

You can use Terraform to codify the configuration for software defined networks. Terraform can then use this configuration to automatically set up and modify settings by interfacing with the control layer. This allows the configuration to be versioned and changes to be automated. For example, you can [use Terraform to configure AWS VPC](https://registry.terraform.io/providers/hashicorp/aws/latest/docs/resources/vpc).

**[»](https://www.terraform.io/intro/use-cases.html" \l "resource-schedulers)Resource Schedulers**

In large-scale infrastructures, static assignment of applications to machines becomes increasingly challenging.

Terraform is not limited to physical providers like AWS. Resource schedulers can be treated as a provider, enabling Terraform to request resources from them. This allows Terraform to be used in layers: to setup the physical infrastructure running the schedulers as well as provisioning onto the scheduled grid.

**[»](https://www.terraform.io/intro/use-cases.html" \l "multi-cloud-deployment)Multi-Cloud Deployment**

Realizing multi-cloud deployments can be very challenging as many existing tools for infrastructure management are cloud-specific (such as cloudformation). Terraform is cloud-agnostic and allows a single configuration to be used to manage multiple providers, and to even handle cross-cloud dependencies. This simplifies management and orchestration, helping operators build large-scale multi-cloud infrastructures.

 Terraform has several advantages over manually managing your infrastructure:

* Terraform can manage infrastructure **on multiple cloud platforms.**
* **The human-readable configuration** language helps you write infrastructure code quickly.
* Terraform's state allows you to **track resource changes throughout** your deployments.
* You can commit your configurations to version control **to safely collaborate on infrastructure.**

To deploy infrastructure with Terraform:

* **Scope** - Identify the infrastructure for your project.
* **Author** - Write the configuration for your infrastructure.
* **Initialize** - Install the plugins Terraform needs to manage the infrastructure.
* **Plan** - Preview the changes Terraform will make to match your configuration.
* **Apply** - Make the planned changes.

**Basic commands :**

1. **terraform init-** initialize the directory
2. **terraform plan :** to see how terraform plans to create infra and see for any changes that are required for your infrastructure
3. **terraform apply:** to create resource in current working directory
4. **terraform destroy:** to destroy resources in current working directory

**Terraform Language**

The Terraform language is declarative, describing an intended goal rather than the steps to reach that goal.

**Configuration Syntax**

Terraform language is defined in terms of a syntax called HCL(HashiCorp Configuration Language), which is also used by configuration languages in other applications, and in particular other HashiCorp products.

The Terraform language syntax is built around two key syntax constructs: arguments and blocks.

An***argument*** assigns a value to a particular name

A***block***is a container for other content

**Identifiers**

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.

Block label can be zero or more. Ex: zero if u use block locals{}

Syntax:

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

# Block body

**<**IDENTIFIER**>** **=** **<**EXPRESSION**>** # Argument

}

**File Extension**

Code in the Terraform language is stored in plain text files with the .tf file extension. There is also [a JSON-based variant of the language](https://www.terraform.io/docs/language/syntax/json.html) that is named with the .tf.json file extension.

Files containing Terraform code are often called configuration files.

## Comments

The Terraform language supports three different syntaxes for comments:

* [#](https://www.terraform.io/docs/language/syntax/configuration.html) begins a single-line comment, ending at the end of the line.
* [//](https://www.terraform.io/docs/language/syntax/configuration.html" \l "-1) also begins a single-line comment, as an alternative to #.
* [/\*](https://www.terraform.io/docs/language/syntax/configuration.html" \l "-2) and \*/ are start and end delimiters for a comment that might span over multiple lines.

**Formatting Rules**

Terraform language also has some idiomatic style conventions which:

* Indent two spaces for each nesting level.
* at the same nesting level, align their equals signs
* When both arguments and blocks appear together inside a block body, place all of the arguments together at the top and then place nested blocks below them. Use one blank line to separate the arguments from the blocks.
* Use empty lines to separate logical groups of arguments within a block.
* For blocks that contain both arguments and "meta-arguments" (as defined by the Terraform language semantics), list meta-arguments first and separate them from other arguments with one blank line.
* Place meta-argument blocks *last* and separate them from other blocks with one blank line.
* Top-level blocks should always be separated from one another by one blank line. Nested blocks should also be separated by blank lines, except when grouping together related blocks of the same type (like multiple provisioner blocks in a resource).
* Avoid separating multiple blocks of the same type with other blocks of a different type, unless the block types are defined by semantics to form a family. (For example: root\_block\_device, ebs\_block\_device and ephemeral\_block\_device on aws\_instance form a family of block types describing AWS block devices, and can therefore be grouped together and mixed.)

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.

# Override Files

Terraform has special handling of any configuration file whose name ends in \_override.tf or \_override.tf.json. This special handling also applies to a file named literally override.tf or override.tf.json.

Terraform initially skips these override files when loading configuration, and then afterwards processes each one in turn (in lexicographical order). For each top-level block defined in an override file, Terraform attempts to find an already-defined object corresponding to that block and then merges the override block contents into the existing object.

Use override files only in special circumstances. Over-use of override files hurts readability

If you have a Terraform configuration example.tf with the following contents:

resource "aws\_instance" "web" {

instance\_type **=** "t2.micro"

ami **=** "ami-408c7f28"

}

...and you created a file override.tf containing the following:

resource "aws\_instance" "web" {

ami **=** "foo"

}

Terraform will merge the latter into the former, behaving as if the original configuration had been as follows:

resource "aws\_instance" "web" {

instance\_type **=** "t2.micro"

ami **=** "foo"

}

If more than one override file defines the same top-level block, the overriding effect is compounded, with later blocks taking precedence over earlier blocks. Overrides are processed in order first by filename (in lexicographical order) and then by position in each file.

The depends\_on meta-argument may not be used in override blocks, and will produce an error for merging in Data , resource & Ouput blocks.

# **Resource Blocks**

 Applying a Terraform configuration will:

* *Create* resources that exist in the configuration but are not associated with a real infrastructure object in the state.
* *Destroy* resources that exist in the state but no longer exist in the configuration.
* *Update in-place* resources whose arguments have changed. Denoted by ~ in plan
* *Destroy and re-create* resources whose arguments have changed but which cannot be updated in-place due to remote API limitations.

This general behavior applies for all resources, regardless of type.

**Operation Timeouts**

* Some resource types provide a special timeouts nested block argument that allows you to customize how long certain operations are allowed to take before being considered to have failed. For example, [aws\_db\_instance](https://registry.terraform.io/providers/hashicorp/aws/latest/docs/resources/db_instance) allows configurable timeouts for create, update and delete operations.

resource "aws\_db\_instance" "example" {

timeouts {

create **=** "60m"

delete **=** "2h"

}}

**Local-only resource** types exist for [generating private keys](https://registry.terraform.io/providers/hashicorp/tls/latest/docs/resources/private_key), [issuing self-signed TLS certificates](https://registry.terraform.io/providers/hashicorp/tls/latest/docs/resources/self_signed_cert), and even [generating random ids](https://registry.terraform.io/providers/hashicorp/random/latest/docs/resources/id). While these resource types often have a more marginal purpose than those managing "real" infrastructure objects, they can be useful as glue to help connect together other resources.

The behavior of local-only resources is the same as all other resources, but their result data exists only within the Terraform state. "Destroying" such a resource means only to remove it from the state, discarding its data.

# **Meta-Arguments**

The Terraform language defines several meta-arguments, which can be used with any resource type to change the behavior of resources.Meta Arguments are used inside Resource Blocks and are feature of Resource

The following meta-arguments are documented on separate pages:

* [depends\_on, for specifying hidden dependencies](https://www.terraform.io/docs/language/meta-arguments/depends_on.html)

depends\_on **=** [aws\_iam\_role\_policy.**example**, ]

* [count, for creating multiple resource instances according to a count](https://www.terraform.io/docs/language/meta-arguments/count.html)
* [for\_each, to create multiple instances according to a map, or set of strings](https://www.terraform.io/docs/language/meta-arguments/for_each.html)
* [provider, for selecting a non-default provider configuration](https://www.terraform.io/docs/language/meta-arguments/resource-provider.html)
* [lifecycle, for lifecycle customizations](https://www.terraform.io/docs/language/meta-arguments/lifecycle.html)
* [provisioner and connection, for taking extra actions after resource creation](https://www.terraform.io/docs/language/resources/provisioners/index.html)

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

resource "azurerm\_resource\_group" "rg" {

for\_each **=** {

a\_group **=** "eastus"

another\_group **=** "westus2"

}

name **=** each.**key**

location **=** each.**value**

}

In blocks where for\_each is set, an additional each object is available in expressions, so you can modify the configuration of each instance. This object has two attributes:

1. [each.key](https://www.terraform.io/docs/language/meta-arguments/for_each.html" \l "each-key) — The map key (or set member) corresponding to this instance.
2. [each.value](https://www.terraform.io/docs/language/meta-arguments/for_each.html" \l "each-value) — The map value corresponding to this instance. (If a set was provided, this is the same as each.key.)

**Lifecycle block**

Resourec behavior can be customized using the special nested lifecycle block within a resource block body:

resource "azurerm\_resource\_group" "example" {

# ...

lifecycle {

create\_before\_destroy **=** **true**

}

}

The following arguments can be used within a lifecycle block:

1. [create\_before\_destroy](https://www.terraform.io/docs/language/meta-arguments/lifecycle.html" \l "create_before_destroy) (bool) : The create\_before\_destroy meta-argument changes default behavior so that the new replacement object is created *first,* and the prior object is destroyed after the replacement is created.
2. [prevent\_destroy](https://www.terraform.io/docs/language/meta-arguments/lifecycle.html" \l "prevent_destroy) (bool) - This meta-argument, when set to true, will cause Terraform to reject with an error any plan that would destroy the infrastructure object associated with the resource, as long as the argument remains present in the configuration.
3. [ignore\_changes](https://www.terraform.io/docs/language/meta-arguments/lifecycle.html" \l "ignore_changes) (list of attribute names) : The ignore\_changes feature is intended to be used when a resource is created with references to data that may change in the future, but should not affect said resource after its creation.

# **Provisioners**

Provisioners can be used to model specific actions on the local machine or on a remote machine in order to prepare servers or other infrastructure objects for service.

### **The self Object**

Expressions in provisioner blocks cannot refer to their parent resource by name. Instead, they can use the special self object.

The self object represents the provisioner's parent resource, and has all of that resource's attributes. For example, use self.public\_ip to reference an aws\_instance's public\_ip attribute.

# **Provisioner Connection Settings**

Most provisioners require access to the remote resource via SSH or WinRM, and expect a nested connection block with details about how to connect.

**Note:** Provisioners should only be used as a last resort. For most common situations there are better alternatives.

Connection blocks don't take a block label, and can be nested within either a resource or a provisioner.

* A connection block nested directly within a resource affects all of that resource's provisioners.
* A connection block nested in a provisioner block only affects that provisioner, and overrides any resource-level connection settings.

One use case for providing multiple connections is to have an initial provisioner connect as the root user to set up user accounts, and have subsequent provisioners connect as a user with more limited permissions:

provisioner "file" {

source **=** "conf/myapp.conf"

destination **=** "/etc/myapp.conf"

connection {

type **=** "ssh"

user **=** "root"

password **=** "${var.root\_password}"

host **=** "${var.host}"

}

}

# Copies the file as the Administrator user using WinRM

provisioner "file" {

source **=** "conf/myapp.conf"

destination **=** "C:/App/myapp.conf"

connection {

type **=** "winrm"

user **=** "Administrator"

password **=** "${var.admin\_password}"

host **=** "${var.host}"

}

}

# Provisioners Without a Resource

If you need to run provisioners that aren't directly associated with a specific resource, you can associate them with a null\_resource.

Instances of [null\_resource](https://registry.terraform.io/providers/hashicorp/null/latest/docs/resources/resource) are treated like normal resources, but they don't do anything. Like with any other resource, you can configure [provisioners](https://www.terraform.io/docs/language/resources/provisioners/syntax.html) and [connection details](https://www.terraform.io/docs/language/resources/provisioners/connection.html) on a null\_resource. You can also use its triggers argument and any meta-arguments to control exactly where in the dependency graph its provisioners will run.

resource "aws\_instance" "cluster" {

count **=** 3}

resource "null\_resource" "cluster" {

# Changes to any instance of the cluster requires re-provisioning

triggers **=** {

cluster\_instance\_ids **=** "${join(",", aws\_instance.cluster.\*.id)}"

}

connection {

host **=** "${element(aws\_instance.cluster.\*.public\_ip, 0)}"

}

provisioner "remote-exec" {

inline **=** [

"bootstrap-cluster.sh ${join(" ", aws\_instance.cluster.\*.private\_ip)}",

]

}

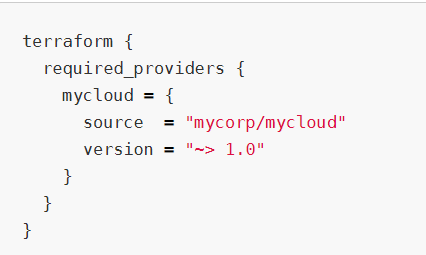
}

**Providers:**

Terraform relies on plugins called "providers" to interact with cloud providers, SaaS providers, and other APIs.

Terraform configurations must declare which providers they require so that Terraform can install and use them. Additionally, some providers require configuration (like endpoint URLs or cloud regions) before they can be used.

* Provider Requirements



Each argument in the required\_providers block enables one provider.

* **[source](https://www.terraform.io/docs/language/providers/requirements.html" \l "source) -** the global [source address](https://www.terraform.io/docs/language/providers/requirements.html" \l "source-addresses) for the provider you intend to use, such as hashicorp/aws.

A provider's source address is its global identifier. It also specifies the primary location where Terraform can download it.

Source addresses consist of three parts delimited by slashes (/), as follows:

**[<HOSTNAME>/]<NAMESPACE>/<TYPE>**

* **Hostname** (optional): The hostname of the Terraform registry that distributes the provider. If omitted, this defaults to registry.terraform.io, the hostname of [the public Terraform Registry](https://registry.terraform.io/).
* **Namespace:** An organizational namespace within the specified registry
* **Type:** A short name for the platform or system the provider manages. Must be unique within a particular namespace on a particular registry host.

**Note:** If you omit the source argument when requiring a provider, Terraform uses an implied source address of registry.terraform.io/hashicorp/<LOCAL NAME>. This is a backward compatibility feature to support the transition to Terraform 0.13; in modules that require 0.13 or later, we recommend using explicit source addresses for all providers.

* **[version](https://www.terraform.io/docs/language/providers/requirements.html" \l "version) -** a [version constraint](https://www.terraform.io/docs/language/providers/requirements.html" \l "version-constraints) specifying which subset of available provider versions the module is compatible with.

Each provider has two identifiers:

* **A unique *source address,*** which is only used when requiring a provider.
* **A *local name,*** which is used everywhere else in a Terraform module.
* Local names must be unique per-module.

Outside of the required\_providers block, Terraform configurations always refer to providers by their local names. For example, the following configuration declares mycloud as the local name for mycorp/mycloud, then uses that local name when [configuring the provider](https://www.terraform.io/docs/language/providers/configuration.html):



* Users of a provider can choose any local name for it. However, nearly every provider has a *preferred local name,* which it uses as a prefix for all of its resource types. (For example, resources from hashicorp/aws all begin with aws, like aws\_instance or aws\_security\_group.)

[**Dependency Lock File**](https://www.terraform.io/docs/language/dependency-lock.html)

* [Version constraints](https://www.terraform.io/docs/language/expressions/version-constraints.html) within the configuration itself determine which versions of dependencies are potentially compatible, but after selecting a specific version of each dependency Terraform remembers the decisions it made in a dependency lock file so that it can (by default) make the same decisions again in future.
* At present, the dependency lock file tracks only provider dependencies. Terraform does not remember version selections for remote modules, and so Terraform will always select the newest available module version that meets the specified version constraints. You can use an exact version constraint to ensure that Terraform will always select the same module version.
* The lock file is always named .terraform.lock.hcl, and this name is intended to signify that it is a lock file for various items that Terraform caches in the .terraform subdirectory of your working directory.

# **Data Sources**

Data sources allow Terraform use information defined outside of Terraform, defined by another separate Terraform configuration, or modified by functions.

When distinguishing from data resources, the primary kind of resource (as declared by a resource block) is known as a managed resource. Both kinds of resources take arguments and export attributes for use in configuration, but while managed resources cause Terraform to create, update, and delete infrastructure objects, data resources cause Terraform only to read objects.

## Local-only Data Sources

While many data sources correspond to an infrastructure object type that is accessed via a remote network API, some specialized data sources operate only within Terraform itself, calculating some results and exposing them for use elsewhere.

For example, local-only data sources exist for [rendering templates](https://registry.terraform.io/providers/hashicorp/template/latest/docs/data-sources/file), [reading local files](https://registry.terraform.io/providers/hashicorp/local/latest/docs/data-sources/file), and [rendering AWS IAM policies](https://registry.terraform.io/providers/hashicorp/aws/latest/docs/data-sources/iam_policy_document).

The behavior of local-only data sources is the same as all other data sources, but their result data exists only temporarily during a Terraform operation, and is re-calculated each time a new plan is created.

## Data Resource Dependencies

Data resources have the same dependency resolution behavior [as defined for managed resources](https://www.terraform.io/docs/language/resources/behavior.html" \l "resource-dependencies). Setting the depends\_on meta-argument within data blocks defers reading of the data source until after all changes to the dependencies have been applied.

**NOTE:** **In Terraform 0.12 and earlier**, due to the data resource behavior of deferring the read until the apply phase when depending on values that are not yet known, using depends\_on with data resources will force the read to always be deferred to the apply phase, and therefore a configuration that uses depends\_on with a data resource can never converge. Due to this behavior, we do not recommend using depends\_on with data resources.

* Data resources support [count](https://www.terraform.io/docs/language/meta-arguments/count.html) and [for\_each](https://www.terraform.io/docs/language/meta-arguments/for_each.html) meta-arguments as defined for managed resources, with the same syntax and behavior.
* Data resources support [the provider meta-argument](https://www.terraform.io/docs/language/meta-arguments/resource-provider.html) as defined for managed resources, with the same syntax and behavior.
* Data resources do not currently have any customization settings available for their lifecycle, but the lifecycle nested block is reserved in case any are added in future versions.
* As data sources are essentially a read only subset of resources, they also support the same [meta-arguments](https://www.terraform.io/docs/language/resources/syntax.html" \l "meta-arguments) of resources with the exception of the [lifecycle configuration block](https://www.terraform.io/docs/language/meta-arguments/lifecycle.html).

# **Module Blocks**

# **Calling a Child Module**

To call a module means to include the contents of that module into the configuration with specific values for its [input variables](https://www.terraform.io/docs/language/values/variables.html). Modules are called from within other modules using module blocks:

module "servers" {

source **=** "./app-cluster"

servers **=** 5

}

A module that includes a module block like this is the calling module of the child module.

The label immediately after the module keyword is a local name, which the calling module can use to refer to this instance of the module.

Within the block body (between { and }) are the arguments for the module. Module calls use the following kinds of arguments:

* The source argument is mandatory for all modules.
* The version argument is recommended for modules from a registry.
* Most other arguments correspond to [input variables](https://www.terraform.io/docs/language/values/variables.html) defined by the module. (The servers argument in the example above is one of these.)
* Terraform defines a few other meta-arguments that can be used with all modules, including for\_each and depends\_on. Also supports- count &provider.

The [taint command](https://www.terraform.io/docs/cli/commands/taint.html) can be used to taint specific resources within a module:

$ terraform taint module.salt\_master.aws\_instance.salt\_master

It is not possible to taint an entire module. Instead, each resource within the module must be tainted separately.

The module installer supports installation from a number of different source types, as listed below.

* [Local paths](https://www.terraform.io/docs/language/modules/sources.html" \l "local-paths)
* [Terraform Registry](https://www.terraform.io/docs/language/modules/sources.html" \l "terraform-registry)
* [GitHub](https://www.terraform.io/docs/language/modules/sources.html" \l "github)
* [Bitbucket](https://www.terraform.io/docs/language/modules/sources.html" \l "bitbucket)
* Generic [Git](https://www.terraform.io/docs/language/modules/sources.html" \l "generic-git-repository), [Mercurial](https://www.terraform.io/docs/language/modules/sources.html" \l "generic-mercurial-repository) repositories
* [HTTP URLs](https://www.terraform.io/docs/language/modules/sources.html" \l "http-urls)
* [S3 buckets](https://www.terraform.io/docs/language/modules/sources.html" \l "s3-bucket)
* [GCS buckets](https://www.terraform.io/docs/language/modules/sources.html" \l "gcs-bucket)

# **Expressions**

# **Type Conversion**

Where possible, Terraform automatically converts values from one type to another in order to produce the expected type. If this isn't possible, Terraform will produce a type mismatch error and you must update the configuration with a more suitable expression.

Terraform automatically converts number and bool values to strings when needed. It also converts strings to numbers or bools, as long as the string contains a valid representation of a number or bool value.

**Heredoc Strings**

Terraform also supports a "heredoc" style of string literal inspired by Unix shell languages, which allows multi-line strings to be expressed more clearly.

**<<**EOT

hello

world

EOT

**Generating JSON or YAML**

Don't use "heredoc" strings to generate JSON or YAML. Instead, use [the jsonencode function](https://www.terraform.io/docs/language/functions/jsonencode.html) or [the yamlencode function](https://www.terraform.io/docs/language/functions/yamlencode.html) so that Terraform can be responsible for guaranteeing valid JSON or YAML syntax.

example **=** jsonencode({

a **=** 1

b **=** "hello"

})

### **Interpolation**

A ${ ... } sequence is an *interpolation,* which evaluates the expression given between the markers, converts the result to a string if necessary, and then inserts it into the final string:

"Hello, ${var.name}!"

In the above example, the named object var.name is accessed and its value inserted into the string, producing a result like "Hello, Juan!".

**Directives**

A %{ ... } sequence is a *directive*, which allows for conditional results and iteration over collections, similar to conditional and for expressions.

The following directives are supported:

* The %{if <BOOL>}/%{else}/%{endif} directive chooses between two templates based on the value of a bool expression:
* "Hello, %{ if var.name != "" }${var.name}%{ else }unnamed%{ endif }!"

The else portion may be omitted, in which case the result is an empty string if the condition expression returns false.

**References to Named Values**

Types of Named Values

The main kinds of named values available in Terraform are:

* Resources : <RESOURCE TYPE>.<NAME>
* Input variables: var.<NAME>
* Local values: local.<NAME>
* Child module outputs: module.<MODULE NAME> , module.<MODULE NAME>.<OUTPUT NAME>.
* Data sources: data.<DATA TYPE>.<NAME>
* Filesystem and workspace info: [path.module](https://www.terraform.io/docs/language/expressions/references.html" \l "path-module), [path.root](https://www.terraform.io/docs/language/expressions/references.html" \l "path-root), [path.cwd](https://www.terraform.io/docs/language/expressions/references.html" \l "path-cwd) is the filesystem path of the current working directory, [terraform.workspace](https://www.terraform.io/docs/language/expressions/references.html" \l "terraform-workspace) is the name of the currently selected [workspace](https://www.terraform.io/docs/language/state/workspaces.html).
* Block-local values: [count.index](https://www.terraform.io/docs/language/expressions/references.html" \l "count-index), [each.key](https://www.terraform.io/docs/language/expressions/references.html" \l "each-key) / each.value, [self](https://www.terraform.io/docs/language/expressions/references.html" \l "self)

**A *for* expression**

the following expression would produce a tuple of strings with all-uppercase letters:

[**for** s **in** var.**list** : upper(s)]

**FUNCTIONS**

1. **Numeric :** 
   * [abs](https://www.terraform.io/docs/language/functions/abs.html) – returns absolute value
   * [ceil](https://www.terraform.io/docs/language/functions/ceil.html) - greater whole no
   * [floor](https://www.terraform.io/docs/language/functions/floor.html)- samller whole no
   * [log](https://www.terraform.io/docs/language/functions/log.html)
   * [max](https://www.terraform.io/docs/language/functions/max.html)
   * [min](https://www.terraform.io/docs/language/functions/min.html)
   * [parseint](https://www.terraform.io/docs/language/functions/parseint.html)
   * [pow](https://www.terraform.io/docs/language/functions/pow.html)- power
   * [signum](https://www.terraform.io/docs/language/functions/signum.html)- outputs sign as -1,0,+1
2. **String:**

* [chomp](https://www.terraform.io/docs/language/functions/chomp.html) - removes newline characters at the end of a string.
* [format](https://www.terraform.io/docs/language/functions/format.html)
* [formatlist](https://www.terraform.io/docs/language/functions/formatlist.html)
* [indent](https://www.terraform.io/docs/language/functions/indent.html)
* [join](https://www.terraform.io/docs/language/functions/join.html)
* [lower](https://www.terraform.io/docs/language/functions/lower.html)
* [regex](https://www.terraform.io/docs/language/functions/regex.html)
* [regexall](https://www.terraform.io/docs/language/functions/regexall.html)
* [replace](https://www.terraform.io/docs/language/functions/replace.html)
* [split](https://www.terraform.io/docs/language/functions/split.html)
* [strrev](https://www.terraform.io/docs/language/functions/strrev.html)
* [substr](https://www.terraform.io/docs/language/functions/substr.html)
* [title](https://www.terraform.io/docs/language/functions/title.html)
* [trim](https://www.terraform.io/docs/language/functions/trim.html)
* [trimprefix](https://www.terraform.io/docs/language/functions/trimprefix.html)
* [trimsuffix](https://www.terraform.io/docs/language/functions/trimsuffix.html)
* [trimspace](https://www.terraform.io/docs/language/functions/trimspace.html)
* [upper](https://www.terraform.io/docs/language/functions/upper.html)

**3.**