

Terraform Cheat Sheet

We created this Terraform Cheat Sheet initially for students of our [DevOps Bootcamp: Terraform](#). But we're now sharing it with any and all Developers that want to learn and remember some of the key functions and concepts of Terraform, and have a quick reference guide to the basics of Terraform.

Want to download a PDF version of this Terraform Cheat Sheet?

Enter your email below and we'll send it to you 

Enter Your Email Address

[SEND ME THE PDF](#)

Unsubscribe anytime.

If you've stumbled across this cheatsheet and are just starting to learn about Terraform and cloud infrastructure, you've made a great choice! If you want to get into cloud technologies, Terraform is where you should start. It's an open-source Infrastructure as Code (IaC) tool, meaning you can manage your cloud infrastructure using code or configuration files. This has made it very popular and is a great tool to learn if you're interested in [becoming a DevOps Engineer](#).

However, if you're stuck in an endless cycle of YouTube tutorials and want to start provisioning real-world cloud infrastructure, become a professional developer, have fun and actually get hired, then come join the Zero To Mastery Academy.

You'll [learn Terraform](#) from actual industry professionals alongside thousands of students in our private [Discord community](#).

You'll not only learn to become a top 10% DevOps Engineer or SysAdmin by learning advanced topics most courses don't cover. As you learn

each concept you'll also put your knowledge into hands-on practice by building real cloud infrastructure from scratch. By the end you'll be able to provision infrastructure on AWS, secure it, and run EC2 instances and services on those instances.

Just want the cheatsheet? No problem! Please enjoy and if you'd like to submit any suggestions, feel free to email us at support@zerotomastery.io

Contents

Terraform Architecture

Installation

[Windows](#)

[Linux \(Ubuntu\) Package Manager](#)

[macOS Package Manager](#)

Terraform CLI

[HCL Comment Styles](#)

Terraform Providers (Plugins)

[Provider Configuration](#)

Terraform Resources

Terraform Variables

[Declaring Variables](#)

[Assigning Values to Variables](#)

[String Interpolation](#)

[Variable Types](#)

Data Sources

Output Values

Loops

[count](#)[for_each](#)[For Expressions](#)[Splat Expressions](#)[Dynamic Blocks](#)

Conditional Expressions

Terraform Locals

Built-in Functions

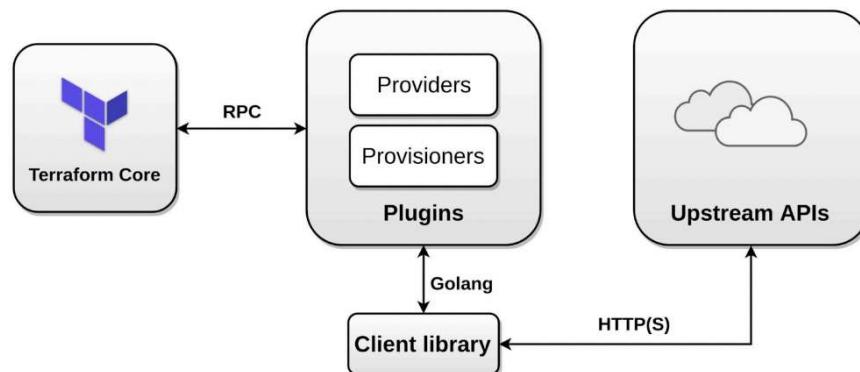
Backends and Remote State

[Backends](#)[Remote State on Amazon S3](#)[Remote State on Terraform Cloud](#)

Terraform Modules

Troubleshooting and Logging

Terraform Architecture



Installation

Windows

1. Download the Windows binary for 32 or 64-bit CPUs from
<https://www.terraform.io/downloads>.
2. Unzip the package.
3. Move the Terraform binary to the Windows PATH.

Linux (Ubuntu) Package Manager

1. Run the following commands at the terminal.

```
1 | curl -fsSL https://apt.releases.hashicorp.com/gpg | sudo apt-key add -
2 | sudo apt-add-repository "deb [arch=amd64] https://apt.releases.hashicorp.com/ $(lsb_release -cs) main"
3 | sudo apt-get update && sudo apt-get install terraform
```



2. Install Terraform using the package manager.

```
1 | sudo apt update && sudo apt install terraform -y
```

macOS Package Manager

Run the following commands at the terminal.

```
1 | brew tap hashicorp/tap
2 | brew install hashicorp/tap/terraform
```

Terraform CLI

`terraform version`

Displays the version of Terraform and all installed plugins.

`terraform -install-autocomplete`

Sets up tab auto-completion, requires logging back in.

`terraform fmt`

Rewrites all Terraform configuration files to a canonical format. Both configuration files (.tf) and variable files (.tfvars) are updated.

Option	Description
<code>-check</code>	Check if the input is formatted. It does not overwrite the file.
<code>-recursive</code>	Also process files in subdirectories. By default, only the given directory (or current directory) is processed.

`terraform validate`

Validates the configuration files for errors. It refers only to the configuration and not accessing any remote services such as remote state, or provider APIs.

`terraform providers`

Prints out a tree of modules in the referenced configuration annotated with their provider requirements.

`terraform init`

Initializes a new or existing Terraform working directory by creating initial files, loading any remote state, downloading modules, etc.

This is the first command that should be run for any new or existing Terraform configuration per machine. This sets up all the local data necessary to run Terraform that is typically not committed to version control.

This command is always safe to run multiple times.

Option	Description
<code>-backend=false</code>	Disable backend or Terraform Cloud initialization for this configuration and use what was previously initialized instead.
<code>-reconfigure</code>	Reconfigure a backend, ignoring any saved configuration.
<code>-migrate-state</code>	Reconfigure a backend and attempt to migrate any existing state.
<code>-upgrade</code>	Install the latest module and provider versions allowed within configured constraints, overriding the default behavior of selecting exactly the version recorded in the dependency lockfile.

`terraform plan`

Generates an execution plan, showing what actions will Terraform take to apply the current configuration. This command will not actually perform the planned actions.

Option	Description
<code>-out=path</code>	Write a plan file to the given path. This can be used as input to the "apply" command.
<code>-input=true</code>	Ask for input for variables if not directly set.
<code>-var 'foo=bar'</code>	Set a value for one of the input variables in the root module of the configuration. Use this option more than once to set more than one variable.
<code>-var-file=filename</code>	Load variable values from the given file, in addition to the default files <code>terraform.tfvars</code> and <code>*.auto.tfvars</code> . Use this option more than once to include more than one variable file.
<code>-destroy</code>	Select the "destroy" planning mode, which creates a plan to destroy all objects currently managed by this Terraform configuration instead of the usual behavior.
<code>-refresh-only</code>	Select the "refresh only" planning mode, which checks whether remote objects still match the outcome of the most recent Terraform apply but does not propose any actions to undo any changes made outside of Terraform.
<code>-target=resource</code>	Limit the planning operation to only the given module, resource, or resource instance and all of its dependencies. You can use this option multiple times to include more than one object. This is for exceptional use only.

`terraform apply`

Creates or updates infrastructure according to Terraform configuration files in the current directory.

Option	Description
<code>-auto-approve</code>	Skip interactive approval of plan before applying.
<code>-replace</code>	Force replacement of a particular resource instance using its resource address.
<code>-var 'foo=bar'</code>	Set a value for one of the input variables in the root module of the configuration. Use this option more than once to set more than one variable.

-var- file=filename	Load variable values from the given file, in addition to the default files <code>terraform.tfvars</code> and <code>*.auto.tfvars</code> . Use this option more than once to include more than one variable file.
-parallelism=n	Limit the number of concurrent operations. Defaults to 10.

Examples:

```
1 | terraform apply -auto-approve -var-file=web-prod.tfvars
2 | terraform apply -replace="aws_instance.server"
```

`terraform destroy`

Destroys Terraform-managed infrastructure and is an alias for
`terraform apply -destroy`

Option	Description
<code>-auto-approve</code>	Skip interactive approval before destroying.
<code>-target</code>	Limit the destroying operation to only the given resource and all of its dependencies. You can use this option multiple times to include more than one object.

Example: `terraform destroy -target aws_vpc.my_vpc -auto-approve`

`terraform taint`

Describes a resource instance that may not be fully functional, either because its creation partially failed or because you've manually marked it as such using this command. Subsequent Terraform plans will include actions to destroy the remote object and create a new object to replace it.

`terraform untaint`

Removes that state from a resource instance, causing Terraform to see it as fully-functional and not in need of replacement.

terraform refresh

Updates the state file of your infrastructure with metadata that matches the physical resources they are tracking. This will not modify your infrastructure, but it can modify your state file to update metadata.

terraform workspace

Option	Description
delete	Delete a workspace.
list	List workspaces.
new	Create a new workspace.
select	Select a workspace.
show	Show the name of the current workspace.

terraform state

This does advanced state management. The state is stored by default in a local file named "**"terraform.tfstate"**", but it can also be stored remotely, which works better in a team environment.

Option	Description
list	List resources in the state.
show	Show a resource in the state.
mv	Move an item in the state.
rm	Remove instances from the state.
pull	Pull current state and output to stdout.

Examples:

```
1  terraform state show aws_instance.my_vm
2  terraform state pull > my_terraform.tfstate
3  terraform state mv aws_iam_role.my_ssm_role
4  terraform state list
5  terraform state rm aws_instance.my_server
```

terraform output

Reads an output variable from a Terraform state file and prints the value. With no additional arguments, output will display all the outputs for the root module.

Examples:

- `terraform output [-json]` : Lists all outputs in the state file.
- `terraform output instance_public_ip` : Lists a specific output value.

`terraform graph`

Produces a representation of the dependency graph between different objects in the current configuration and state. The graph is presented in the DOT language. The typical program that can read this format is GraphViz, but many web services are also available to read this format.

Linux Example:

```
1 | sudo apt install graphviz
2 | terraform graph | dot -Tpng > graph.png
```

`terraform import`

Import existing infrastructure into your Terraform state. This will find and import the specified resource into your Terraform state, allowing existing infrastructure to come under Terraform management without having to be initially created by Terraform.

Example: `terraform import aws_instance.new_server i-123abc`

Imports EC2 instance with id i-abc123 into the Terraform resource named "new_server" of type "aws_instance".

`terraform login [hostname]`

Retrieves an authentication token for the given hostname, if it supports automatic login, and saves it in a credentials file in your home directory. If no hostname is provided, the default hostname is app.terraform.io, to log in to Terraform Cloud.

```
terraform logout [hostname]
```

Removes locally-stored credentials for the specified hostname. If no hostname is provided, the default hostname is app.terraform.io.

HCL Comment Styles

#	single-line comment.
//	single-line comment (alternative to #).
/* ... */	multi-line comment (block comment).

Terraform Providers (Plugins)

A provider is a Terraform plugin that allows users to manage an external API.

A provider usually provides resources to manage a cloud or infrastructure platform, such as AWS or Azure, or technology (for example Kubernetes).

There are providers for Infrastructure as a Service (IaaS), Platform as a Service (PaaS), and Software as a Service (SaaS).

Provider Configuration

```
1  terraform {
2    required_providers {
3      aws = {
4        source  = "hashicorp/aws"          # provider local name
5        version = "~> 3.0"              # global and unique source address
6      }
7    }
8  }
9
10 # Configure the AWS Provider
11 provider "aws" {
12   region = "us-central-1" # provider configuration options
13 }
```

Terraform Resources

Resources are the most important element in the Terraform language. It describes one or more infrastructure objects to manage.

Together the resource **type** and **local name** serve as an identifier for a given resource and must be unique within a module. Example:

```
aws_vpc.main
```

Creating resources:

```
1  resource "<provider>_<resource_type>" "local_name"{
2      argument1 = value
3      argument2  = value
4      ...
5  }
6
7  # Example:
8  resource "aws_vpc" "main" {
9      cidr_block = "10.0.0.0/16"
10     enable_dns_support = true
11
12    tags = {
13        "Name" = "Main VPC"
14    }
15 }
```

Terraform Variables

Input variables allow you customize aspects of Terraform without using hard-coded values in the source.

Declaring a variable

Variable declarations can appear anywhere in your configuration files. However, it's recommended to put them into a separate file called **variables.tf**.

```
1  # variable declaration
2  variable "vpc_cidr_block" {
3      description = "CIDR block for VPC".
4      default = "192.168.0.0/16"
5      type = string
6  }
```

Assigning values to variables

1. Using the **default** argument in the variable declaration block.
2. Assign a value to the variable in the variable definition file which by default is **terraform.tfvars**. Example: `vpc_cidr_block = "172.16.0.0/16"`
3. Using **-var** command-line option. Example: `terraform apply -var="vpc_cidr_block=10.0.10.0/24"`
4. Using **-var-file** command-line option. Example: `terraform apply -auto-approve -var-file=web-prod.tfvars`
5. Exporting the variable at the terminal. Example: `export TF_VAR_vpc_cidr_block="192.168.100.0/24"`

Variable definition precedence (from highest to lowest):

1. Variables specified at the terminal using** `-var**` and **-var-file** options.
2. Variables defined in **terraform.tfvars**.
3. Variables defined as environment variables using **TF_VAR** prefix.

String Interpolation

You can interpolate other values in strings by these values in `${} , such as ${var.foo} .`

The interpolation syntax is powerful and allows you to reference variables, attributes of resources, call functions, etc.

You can escape interpolation with double dollar signs: `$$ {foo}` will be rendered as a literal `${foo} .`

Variable Types

1. **Simple types** a. number b. string c. bool d. null
2. **Complex types** a. Collection types i. list ii. map iii. set b. Structural types i. tuple object

type number

```

1 | variable "web_port" {
2 |   description = "Web Port"
3 |   default = 80
4 |   type = number
5 |

```

type string

```

1 | variable "aws_region" {
2 |   description = "AWS Region"
3 |   type = string
4 |   default = "eu-central-1"
5 |

```

type bool

```

1 | variable "enable_dns" {
2 |   description = "DNS Support for the VPC"
3 |   type = bool
4 |   default = true
5 |

```

type list (of strings)

```

1 | variable "azs" {
2 |   description = "AZs in the Region"
3 |   type = list(string)
4 |   default = [
5 |     "eu-central-1a",
6 |     "eu-central-1b",
7 |     "eu-central-1c"
8 |   ]
9 |

```

type map

```

1 | variable "amis" {
2 |   type = map(string)
3 |   default = {
4 |     "eu-central-1" = "ami-0dcc0ebde7b2e00db",
5 |     "us-west-1" = "ami-04a50faf2a2ec1901"
6 |   }
7 |

```

type tuple

```

1 |   variable "my_instance" {
2 |     type = tuple([string, number, bool])
3 |     default = ["t2.micro", 1, true ]
4 |   }

```

type object

```

1 |   variable "egress_dsg" {
2 |     type = object({
3 |       from_port = number
4 |       to_port = number
5 |       protocol = string
6 |       cidr_blocks = list(string)
7 |     })
8 |     default = {
9 |       from_port = 0,
10 |      to_port = 65365,
11 |      protocol = "tcp",
12 |      cidr_blocks = ["100.0.0.0/16", "200.0.0.0/16", "0.0.0.0/0"
13 |    }
14 |  }

```

Data Sources

Data sources in Terraform are used to get information about resources external to Terraform. For example, the public IP address of an EC2 instance. Data sources are provided by providers.

Use Data Sources

A **data block** requests that Terraform read from a given data source ("aws_ami") and export the result under the given local name ("ubuntu").

The data source and name together serve as an identifier for a given resource and therefore must be unique within a module.

Within the block body (between { and }) are query constraints defined by the data source.

```

1 |   data "aws_ami" "ubuntu" {
2 |     most_recent = true
3 |

```

```

4   owners = ["self"]
5   tags = {
6     Name    = "app-server"
7     Tested  = "true"
8   }
9 }
```

Output Values

Output values print out information about your infrastructure at the terminal, and can expose information for other Terraform configurations (e.g. modules) to use.

Declare an Output Value

Each output value exported by a module must be declared using an **output block**. The label immediately after the output keyword is the name.

```

1   output "instance_ip_addr" {
2     value = aws_instance.server.private_ip
3 }
```

Loops

Terraform offers the following looping constructs, each intended to be used in a slightly different scenario:

- **count** meta-argument: loop over resources.
- **for_each** meta-argument: loop over resources and inline blocks within a resource.
- **for** expressions: loop over lists and maps.

count

The **count** meta-argument is defined by the Terraform language and can be used to manage similar resources.

count is a looping technique and can be used with modules and with every resource type.

```

1   # creating multiple EC2 instances using count
2   resource "aws_instance" "server" {
3 }
```

```

1
2
3
4   ami = "ami-06ec8443c2a35b0ba"
5   instance_type = "t2.micro"
6   count = 3 # creating 3 resources
7 }
```

In blocks where count is set, an additional count object is available.

count.index represents the distinct index number (starting with 0) corresponding to the current object.

for_each

for_each is another meta-argument used to duplicate resources that are similar but need to be configured differently.

for_each was introduced more recently to overcome the downsides of count.

If your resources 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.

```

1 # declaring a variable
2 variable "users" {
3   type = list(string)
4   default = ["demo-user", "admin1", "john"]
5 }
6
7 # creating IAM users
8 resource "aws_iam_user" "test" {
9   for_each = toset(var.users) # converts a list to a set
10  name = each.key
11 }
```

For Expressions

A **for** expression creates a complex type value by transforming another complex type value.

```

variable "names" {
  type = list
  default = ["daniel", "ada'", "john wick"]
}

output "show_names" {
  # similar to Python's list comprehension
  value = [for n in var.names : upper(n)]
```

```

10    }
11
12    output "short_upper_names" {
13      # filter the resulting list by specifying a condition:
14      value = [for name in var.names : upper(name) if length(name)
15    ]

```

If you run `terraform apply -auto-approve` you'll get:

```

1 Outputs:
2
3 short_upper_names = [
4   "JOHN WICK",
5 ]
6 show_names = [
7   "DANIEL",
8   "ADA",
9   "JOHN WICK",
10 ]

```

Splat Expressions

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

```

1 # Launch an EC2 instance
2 resource "aws_instance" "server" {
3   ami = "ami-05cafdf7c9f772ad2"
4   instance_type = "t2.micro"
5   count = 3
6 }
7
8 output "private_addresses"{
9   value = aws_instance.server[*].private_ip # splat expression
10 }

```

Dynamic Blocks

Dynamic blocks act much like a `for` expression, but produce nested blocks instead of a complex typed value. They iterate over a given complex value, and generate a nested block for each element of that complex value.

They are supported inside **resource**, **data**, **provider**, and **provisioner** blocks.

A dynamic block produces nested blocks instead of a complex typed value. It iterates over a given complex value, and generates a nested block for each element of that complex value.

```

1 # Declaring a variable of type list
2 variable "ingress_ports" {
3     description = "List Of Ingress Ports"
4     type = list(number)
5     default = [22, 80, 110, 143]
6 }
7
8 resource "aws_default_security_group" "default_sec_group" {
9     vpc_id = aws_vpc.main.id
10
11    # Creating the ingress rules using dynamic blocks
12    dynamic "ingress"{
13        for_each = var.ingress_ports # iterating over the list variable
14        iterator = iport
15        content {
16            from_port = iport.value
17            to_port = iport.value
18            protocol = "tcp"
19            cidr_blocks = ["0.0.0.0/0"]
20        }
21    }
22 }
```

Conditional Expressions

A **conditional expression** uses the value of a boolean expression to select one of two values.

Syntax: `condition ? true_val : false_val`

If condition is true then the result is `true_val`. If condition is false then the result is `false_val`.

The condition can be any expression that resolves to a boolean value. This will usually be an expression that uses the equality, comparison, or logical operators.

```

1 variable "istest" {
2     type = bool
3     default = true
4 }
5
6 # Creating the test-server instance if `istest` equals true
7 resource "aws_instance" "test-server" {
8     ami = "ami-05cafdf7c9f772ad2"
```

```

9   instance_type = "t2.micro"
10  count = var.istest == true ? 1:0 # conditional expression
11  }
12
13  # Creating the prod-server instance if `istest` equals false
14  resource "aws_instance" "prod-server" {
15    ami = "ami-05cafdf7c9f772ad2"
16    instance_type = "t2.large" # it's not free tier eligible
17    count = var.istest == false ? 1:0 # conditional expression
18  }

```

Terraform Locals

Terraform **local values** or simply **locals** are named values that you can refer to in your configuration.

Compared to variables, Terraform locals do not change values during or between Terraform runs and unlike input variables, locals are not submitted by users but calculated inside the configuration.

Locals are available only in the current module. They are locally scoped.

```

# the local values are declared in a single `locals` block
locals {
  owner = "DevOps Corp Team"
  project = "Online Store"
  cidr_blocks = ["172.16.10.0/24", "172.16.20.0/24", "172.16.30
  common-tags = {
    Name = "dev"
    Environment = "development"
    Version = 1.10
  }
}

# Create a VPC.
resource "aws_vpc" "dev_vpc" {
  cidr_block = "172.16.0.0/16"
  tags = local.common-tags
}

# Create a subnet in the VPC
resource "aws_subnet" "dev_subnets" {
  vpc_id          = aws_vpc.dev_vpc.id
  cidr_block      = local.cidr_blocks[0]
  availability_zone = "eu-central-1a"

  tags = local.common-tags
}

# Create an Internet Gateway Resource
resource "aws_internet_gateway" "dev_igw" {
  vpc_id = aws_vpc.dev_vpc.id
}

```

```

31
32   tags = {
33     "Name" = "${local.common-tags["Name"]}-igw"
34     "Version" = "${local.common-tags["Version"]}"
35   }
}

```

Note: Local values are created by a locals block (plural), but you reference them as attributes on an object named local (singular).

Built-in Functions

Terraform includes a number of **built-in functions** that can be called from within expressions to transform and combine values.

Examples of functions: min, max, file, concat, element, index, lookup.

Terraform does not support user-defined functions.

There are functions for numbers, strings, collections, file system, date and time, IP Network, Type Conversions and more.

You can experiment with the behavior of Terraform's built-in functions from the Terraform console, by running the `terraform console` command.

Examples:

```

1 > max(5, 12, 9)
2 12
3
4 > min(12, 54, 3)
5 3
6
7 > format("There are %d lights", 4)
8 There are 4 lights
9
10 > join(", ", ["foo", "bar", "baz"])
11 foo, bar, baz
12
13 > split(", ", "foo,bar,baz")
14 [
15   "foo",
16   "bar",
17   "baz",
18 ]
19
20 > replace("hello world", "/w.*d/", "everybody")
21 hello everybody
22

```

```

23 > substr("hello world", 1, 4)
24 ello
25
26 > element(["a", "b", "c"], 1)
27 b
28
29 > lookup({a="ay", b="bee"}, "a", "what?")
30 ay
31 > lookup({a="ay", b="bee"}, "c", "what?")
32 what?
33
34 > slice(["a", "b", "c", "d"], 1, 3)
35 [
36   "b",
37   "c",
38 ]
39
40 > timestamp()
41 "2022-04-02T05:52:48Z"
42
43 > formatdate("DD MMM YYYY hh:mm ZZZ", "2022-01-02T23:12:01Z")
44

```



Academy

Testimonials

Blog

Cheat Sheets

Community

SIGN IN

JOIN ZERO TO MASTERY

```

48
49 > cidrhost("10.1.2.240/28", 14)
50 10.1.2.254

```

Backends and Remote State

Backends

Each Terraform configuration has an associated backend that defines how operations are executed and where the Terraform state is stored.

The default backend is local, and it stores the state as a plain file in the current working directory.

The backend needs to be initialized by running `terraform init`.

If you switch the backend, Terraform provides a migration option which is `terraform init -migrate-state`.

Terraform supports both local and remote backends:

- local (default) backend stores state in a local JSON file on disk.
- remote backends stores state remotely. Examples of remote backends are AzureRM, Consul, GCS, Amazon S3, and Terraform

Cloud. They can support features like remote operation, state locking, encryption, and versioning.

Configure Remote State on Amazon S3

1. On the AWS console go to Amazon S3 and create a bucket.
2. Configure Terraform to use the remote state from within the S3 bucket.

```
1  terraform {  
2      backend "s3" {  
3          bucket = "bucket_name"  
4          key    = "s3-backend.tfstate"  
5          region = "eu-central-1"  
6          access_key = "AKIA56LJEQNM"  
7          secret_key = "0V9cw4CVON2w1"  
8      }  
9  }
```

3. Run `terraform init` to initialize the backend.

Configure Remote State on Terraform Cloud

1. The first step is to [sign up for a free Terraform Cloud account](#).
2. Create your organization or join a new one.
3. Configure Terraform to use the remote state from within the S3 bucket.

```
terraform {  
  required_providers {  
    aws = {  
      source  = "hashicorp/aws"  
      version = "~> 3.0"  
    }  
  }  
  cloud {  
    organization = "master-terraform" # should already exist on  
    workspaces {  
      name = "DevOps-Production"  
    }  
  }  
}
```

```
    }  
}
```

4. Authenticate to Terraform Cloud to proceed with initialization.
5. Run 'terraform login'.
6. Run 'terraform init' to initialize the backend.

Terraform Modules

Terraform modules are a powerful way to reuse code and stick to the **DRY principle**, which stands for "Do Not Repeat Yourself". Think of modules as functions in a programming language.

Modules will help you organize configuration, encapsulate configuration, re-use configuration and provide consistency and ensure best-practices.

Terraform supports Local and Remote modules:

- **Local modules** are stored locally, in a separate directory, outside of the root environment and have the source path prefixed with `./` or `../`
- **Remote modules** are stored externally in a separate repository, and support versioning. External Terraform modules are found on the [Terraform Registry](#).

A Terraform module is a set of Terraform configuration files in a single directory.

When you run Terraform commands like `terraform plan` or `terraform apply` directly from such a directory, then that directory will be considered the root module.

The modules that are imported from other directories into the root module are called **child modules**.

Calling a child module from within the root module:

```
1 | module "myec2" {  
2 |   # path to the module's directory  
3 |   # the source argument is mandatory for all modules.  
4 |   source = ".../modules/ec2"  
5 | }
```

```
5
6  # module inputs
7  ami_id = var.ami_id
8  instance_type = var.instance_type
9  servers = var.servers
10 }
```

It's good practice to start building everything as a module, create a library of modules to share with your team and from the very beginning to start thinking of your entire infrastructure as a collection of reusable modules.

After adding or removing a module, you must re-run `terraform init` to install the module.

Troubleshooting and Logging

The **TF_LOG** enables logging and can be set to one of the following log levels: TRACE, DEBUG, INFO, WARN or ERROR.

Once you have configured your logging you can save the output to a file. This is useful for further inspection.

The **TF_LOG_PATH** variable will create the specified file and append the logs generated by Terraform.

Example:

```
1  export TF_LOG_PATH=terraform.log
2  terraform apply
```

You can generate logs from the core application and the Terraform provider separately.

To enable core logging, set the **TF_LOG_CORE** environment variable, and to generate provider logs set the **TF_LOG_PROVIDER** to the appropriate log level.

[Pricing](#)[Career Paths](#)[Swag Store](#)[Testimonials](#)[Workshops &](#)[Ambassadors](#)[Blog](#)[More](#)[Contact Us](#)[Cheat Sheets](#)[Career Path](#)[Newsletters](#)[Quiz](#)[Community](#)[Free Resources](#)[Privacy](#) [Terms](#) [Cookies](#)**Copyright © 2024, Zero To Mastery Inc.**