Section 6

Managing and Optimizing your Terraform configuration

Understanding Arguments and Attributes

The Terraform block syntax uses arguments and attributes to create the configuration for resources created in an infrastructure provider

Arguments assigns a value to a name mostly in a resource block. The name being a resource configuration parameter and the value being expressed in many ways by Terraform.

Attributes are a piece of data associated with a resources like its name, id, ip address, DNS name etc...

Terraform expressions and their value types

In programming, expressions are used to evaluate a syntax in a programming language in order to determine its value. This syntax can be constants, arithmetic operations, functions, variables, etc...

expression:

result = 1 + 2 - 12 / 5 *... or “red” or true/false

Terraform uses expressions in the same way to evaluate the expression syntax to determine its value and assign it to an argument name.

Expression can be literal values, input variables, built-in functions, conditional evaluation, local values, arithmetics (a+b) or reference to another resource attribute,...

Value can have various types

- Primitive type is one piece of data as the value.
- Collection type groups multiple pieces of data into one value.

The primitive types for values can be:

Strings:
letters like hello or red using quotes “ ”
numbers:
Any numbers including with decimals
Boolean:
True or false.

Null : to be ignored

Collection of data of the same type:

- list: [“test” , “test2”]

ordered and identifiable using a zero based placement number..known as arrays

- Map: { “age”=15, “age2”=20 } <key=value>

ordered and identifiable using a zero based numbering system.

- set : (“test”, “red”, “long”)

not ordered and not identifiable

Collection of data of a different data types:

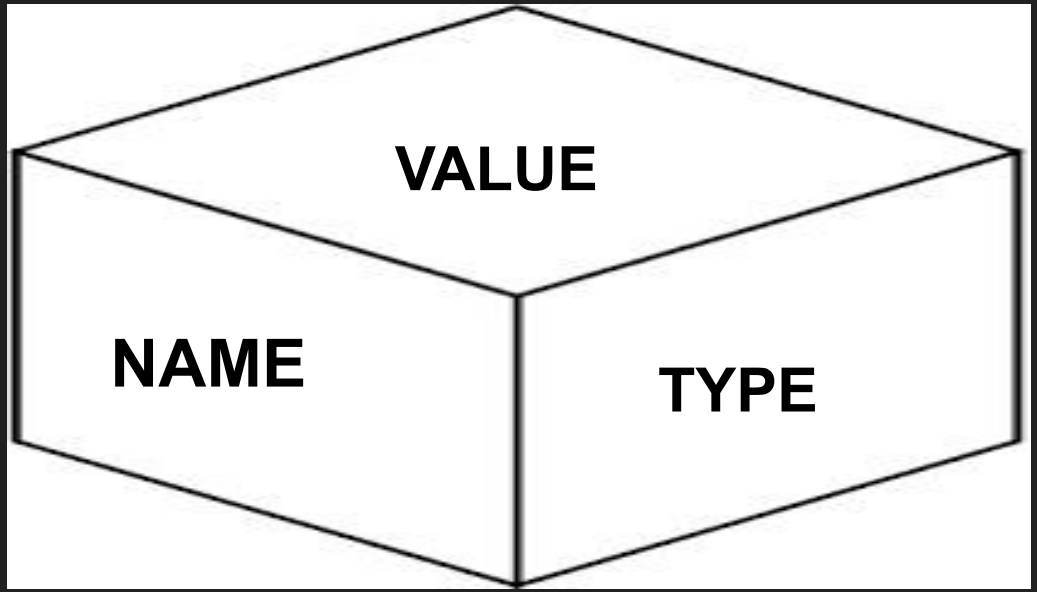
- object: map of different date types { “age”=20 , “height”=6 }
- tuple : list of different data types [“papa”, 4 , false]

Using Terraform variables or input variables..

In programing, Variables are important and assigns a name for the values that can change during the execution of a program

we can change the value of the variable in one central location without having to change the code.

VARIABLE = NAME * VALUE * TYPE



Terraform variables are used in a config block to represent the value of an expression with the syntax:

var.<variable_name>

The input variables are configured using a variable configuration block and organized in a variable.tf file.

If not defined, variable values can be passed inline, with the -var flag or as a file with -var-file with terraform plan and apply commands.

Variable configuration block
default value argument will assign the value

description argument will help understand the role of the input variable.

Type argument will specify the type of value...string, number, boolean, list etc...

Input variables have a dynamic type or type any when the type is not yet known.

Input variables can also be configured as environment variables instead of variable config block

Environment variables are set using the command

export TF_VAR_VARIABLE_NAME="value"

ie:

export TF_VAR_AWS_ACCESS_KEY_ID="AXUTYUISQAAA"

export TF_VAR_AWS_SECRET_ACCESS_KEY="cH567ujKIPPPO90"

Non default or values for each of the input variables configured in the variable.tf file can be declared in a terraform.tfvars configuration file.

The file name terraform.tfvars has to be used to be recognized by Terraform.

Other names can be used but it will have to passed to a terraform command using the -var-file flag.

Variables CANNOT use the following names:

- Source
- Count
- For_each
- Version
- Provider
- Lifecycle
- Locals
- depends_on

Output values in a root module

Terraform output values will return values of any data related to a resource provisioned in a provider to the CLI after a terraform apply is successfully completed.

The value can actually be anything you define ..more practically... argument or an attribute of a provisioned resource

Syntax ... value = <resource_type>.<name>.<argument_name>
<resource_type>.<name>.<attribute>

<resource_type>.<name> represent TF managed resource

Attribute >> data assigned by provider when resource is created as provider id, ip addresses, dns names etc..

Output values are configured with the output block to define the value that you want to output to the CLI.

recommends to be organized output configs in an output.tf configuration file.

Output values can be viewed with the $ terraform output command by reading the state file output{} field. Need refresh or apply for added config.

Using Data Sources in Terraform

A Data source is the location of the data being used originates from...

Two types of TF resources: data resource and managed resources..Data sources are accessed via the Terraform data resource

Terraform data resource is a read only operation to fetch infrastructure provider resources data

Each provider will have its own set of data sources which are accessed by Terraform via the API

Terraform data resources are configured or declared using a data configuration block.

Data sources can be used as expressions in resource blocks with the syntax:

data.<data_resource_type><name>.attribute

How to Refer to resource attributes in an expression

Attributes == infrastructure provider metadata

Metadata == piece of information about a resource

Infrastructure providers assign attributes to all their resources. Some are assigned only when resources is provisioned.

For example.. resource id or resource number or IP addresses or DNS name

These Resource attributes values can be used in other resource block as expressions using the following reference syntax:

<resource_type>.<resource_name>.<attribute>

Note: This will create an implicit dependency between the 2 resources.

Scaling resources with count and for_each.

You will have to create multiple instances of resources of the same type in an infrastructure provider like servers, networks, load balancers, subnets etc..

With Terraform that would mean duplicating the same configuration block for each resource of the same type for n amount of times...

Terraform solves this problem with the count and for_each Meta arguments with resource or data blocks.

With count, Terraform assigns a 0 based numerical index number to each resource created to uniquely identify each of the resources provisioned in the execution plan.

With count, Terraform adds the index number to each managed resource address.
The resource address now becomes the list of all resources created with the count value:

<resource_type><name>[index] (index=0,1,2,3..)

count.index can be used to look up the index value and modify an expression value based on the index number.

Numbers are not always the best way to address resources and what if expression value are distinct??

for_each allows to append use a more descriptive identifier with a string to each resource using maps or sets to assign distinct values to each resource.

each.key and each.value are used as expression using the keys and values defined in for_each.

```
resource “resource_type” “name”{

    for each { “key1”=”value1”,....}

   argument = each.key
   ...
   argument = each.value
  ...
}
```

Each resource provisioned will iterate over the map or set. For maps, expressions will use key and value. With sets, key and value are the same.

Terraform Resource addressing

The address for Terraform managed resources uses the following syntax

<resource_type>.<resource_name>

With count and for_each, the bracketed index can be added to address a particular resource.

The bracketed 0-based numerical or string index can also be used to address the value of an element of a list or a map when used in an expression.

When using for_each the index refers to a particular resource using the key or the value within the map or set defined in the for_each block

<resource_type><name>.[“index”]

**Index is a string

This can be used for data sources as well:

data.<data_resource_type><name>.attribute[index]

Leveraging the power of Terraform Built-in Functions

A function is a block of organized, reusable code that is used to perform a single, related action.

With Terraform, functions are used as a type of expression to evaluate a value for arguments.

Terraform have a lot of useful built-in functions such as cidrsubnet(), file(), element() that can help as your TF config grows more complex.

They can be used in expression with the syntax <function_name>( )

IMPORTANT: Terraform does not support custom-built functions.

Using Terraform Dynamic blocks

In programming, nested code blocks are code blocks within another code block being the parent or top level code block.

In Terraform they are used in a config block for arguments that cannot be assigned an expression but rather a block of code and thus using a nested block instead.

They are used with top level block such as the resource, data resource, provider and provisioner block type

Dynamic blocks allows to dynamically create repeatable nested blocks within the supported top level block types.

It will use for_each to create these repeatable nested blocks.

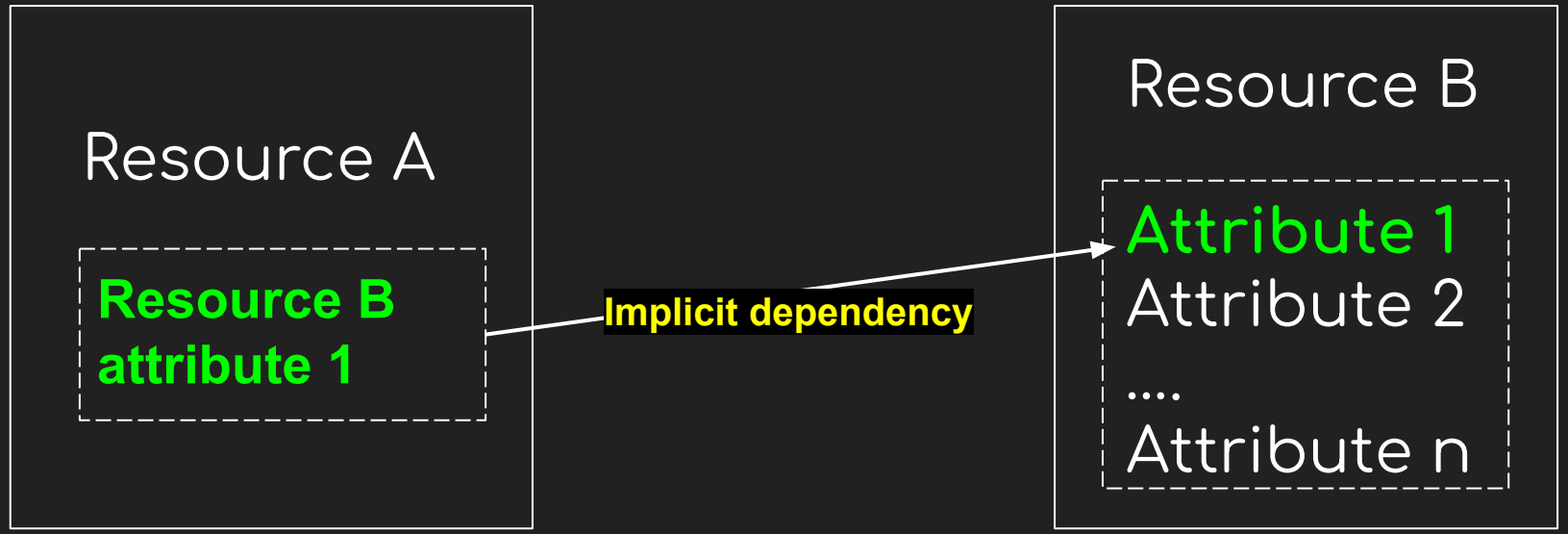
Best practise is to use them in reusable modules and in only when needed.

Terraform Dependencies: Implicit or Explicit

When a resource expression is using another resource data from one or more different resource to evaluate its value like an attribute or the value of an argument , it creates a dependency between the resources.

This will determine the order of resource creation and destruction.

An implicit dependency is created between the resources.



An argument in the Resource A config needs the value of an attribute of resource in order to be created. A is implicitly dependent on B due its config block.

When resources are not implicitly dependent, you can make them explicitly dependent using the depends_on argument in a resource block.

A resource can be explicitly dependent on one or more resource and will be created/destroyed based on the dependency.

If there are NO dependencies then resources will be created in parallel also known as parallelism. The default being 10

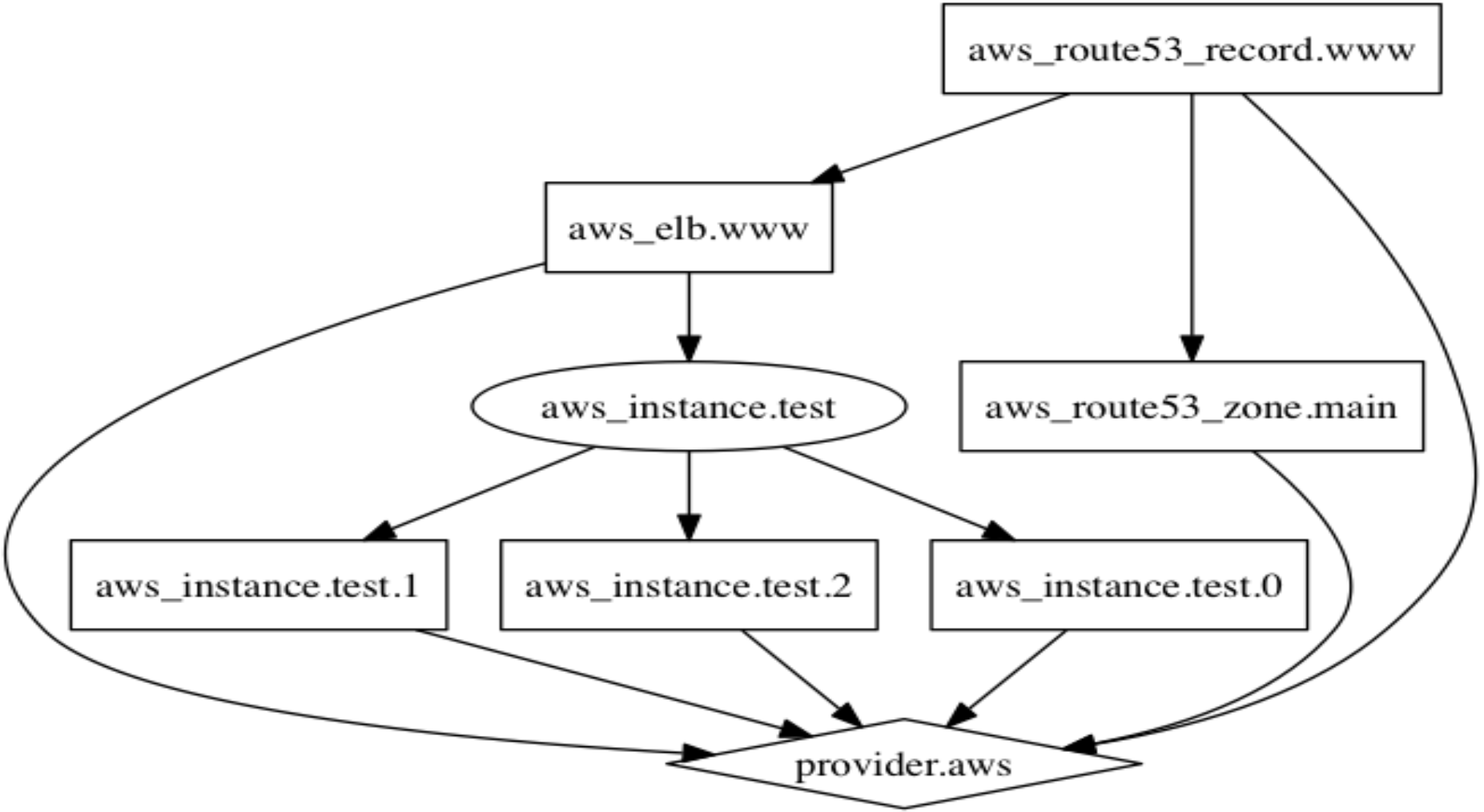
The Terraform graph

Terraform provides a way to visualize the execution and all the dependencies with a dot file or image.
The graph will be automatically generated in the dot format using the terraform graph command but hard to visualize.

$ terraform graph > mygraph.dot

The graph and the dependencies can be visualized as an image with the graphviz utility

$ terraform graph | dot -Tsvg > mygraph.svg


aws_route53_record.www
aws_elb.www
aws_instance.test
aws_route53_zone.main
aws_instance.test.1
aws_instance.test.2
aws_instance.test.0
provider.aws

Debugging Terraform

Debugging can be done by configuring logging with various levels of verbosity or level of log output details.

Terraform logging is enabled with the environment variable TF_LOG with:

$ export TF_LOG=TRACE | INFO | WARN | DEBUG | ERROR

Note: use $setx with windows. Last as long as shell.

TRACE is the most verbose.

```
:14:57 [TRACE] Executing graph transform
*terraform.Remo01vedModuleTransformer
2020/09/06 01:14:57 [TRACE] Completed graph
transform *terraform.RemovedModuleTransformer (no
changes)
2020/09/06 01:14:57 [TRACE] Executing graph
transform *terraform.AttachSchemaTransformer
```

[WARN] Log levels other than TRACE are currently unreliable, and are supported only for backward compatibility.
Use TF_LOG=TRACE to see Terraform's internal logs.

Crashing Terraform

```
2019-09-17T15:29:40.866+0100 [DEBUG] plugin: plugin exited



!!!!!!!!!!!!!!!!!!!!!!!!!!!! TERRAFORM CRASH !!!!!!!!!!!!!!!!!!!!!!!!!!!!

Terraform crashed! This is always indicative of a bug within Terraform.
A crash log has been placed at "crash.log" relative to your current
working directory. It would be immensely helpful if you could please
report the crash with Terraform[1] so that we can fix this.

When reporting bugs, please include your terraform version. That
information is available on the first line of crash.log. You can also
get it by running 'terraform --version' on the command line.

[1]: https://github.com/hashicorp/terraform/issues

!!!!!!!!!!!!!!!!!!!!!!!!!!!! TERRAFORM CRASH !!!!!!!!!!!!!!!!!!!!!!!!!!!!
```

Terraform can experience crashes due to a panic in the GO code. It will output all the tracebacks at the cli and save the crash details in a crash.log file saved in the root module.

```
panic: runtime error: invalid memory address or
nil pointer dereference
[signal SIGSEGV: segmentation violation code=0x1
addr=0x8 pc=0x1a0242e]
```

Zero Downtime Deployments (ZDD) with lifecycle

The Lifecycle Meta-argument block in a resource block provides some level of ZDD.

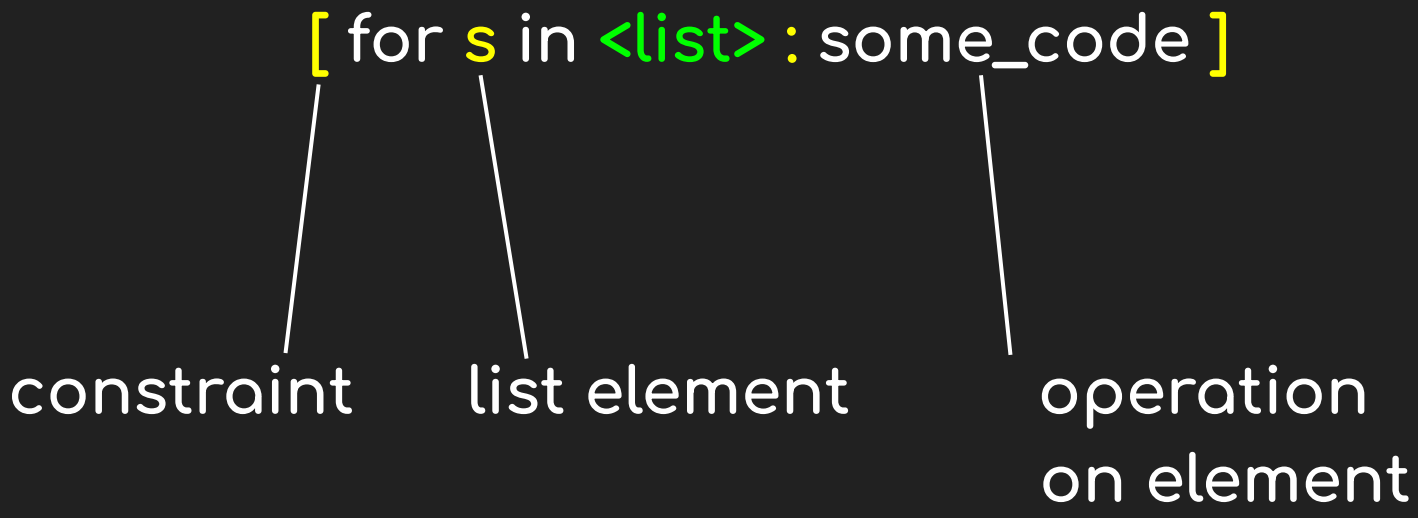
By default, resources that are tainted and updated in some cases will be first destroyed and then recreated which can lead to downtime or outage. -/+

The lifecycle argument will allow to change the order to create_before_destroy . +/-
**Other arguments are prevent_destroy and ignore_changes

The Terraform for and splat expressions

In programming, for is to create a loop to repeat the execution of the same block of code. One use case is to iterate over elements of a list and repeat the same code for each element.

With Terraform, for is used as an expression to iterate over an existing list or a map and perform an action to evaluate an expression..


[for s in <list> : some_code]
constraint
list element
operation
on element

The splat expression or [*] is added to a list and used to iterate over every element of a list.

Used to output the attribute of elements in a list that have attributes ....good use case is when using count.

```
list[*].attribute
```

Not supported with for_each.

Terraform conditional expressions

Conditional expression allows to pick a value based on a condition being true or false.
Conditional expression use a single line instead of multiline if else statements.
The condition can be set using an operator such as equality (=, !=) or comparison (>, <, <=, >=).

```
$ Condition ? true_value : false_value
```

? is known as the ternary operator

Terraform Vault overview

When using providers, API keys and credentials will be required for Terraform to interact with those services. It is critical to secure those credentials and not expose them

Vault is a Terraform provider to secure, store and inject these secrets to configuration and exposes an API to interacts with Terraform like any other provider.

NOTE: Vault will keep the secrets in the state and plan files in clear text.

A token based authentication mechanism is used by Vault to authenticate Terraform in order for it to pass any secrets.

The lifetime of the token can be configured.