HashiCorp
Terraform

Terraform Training

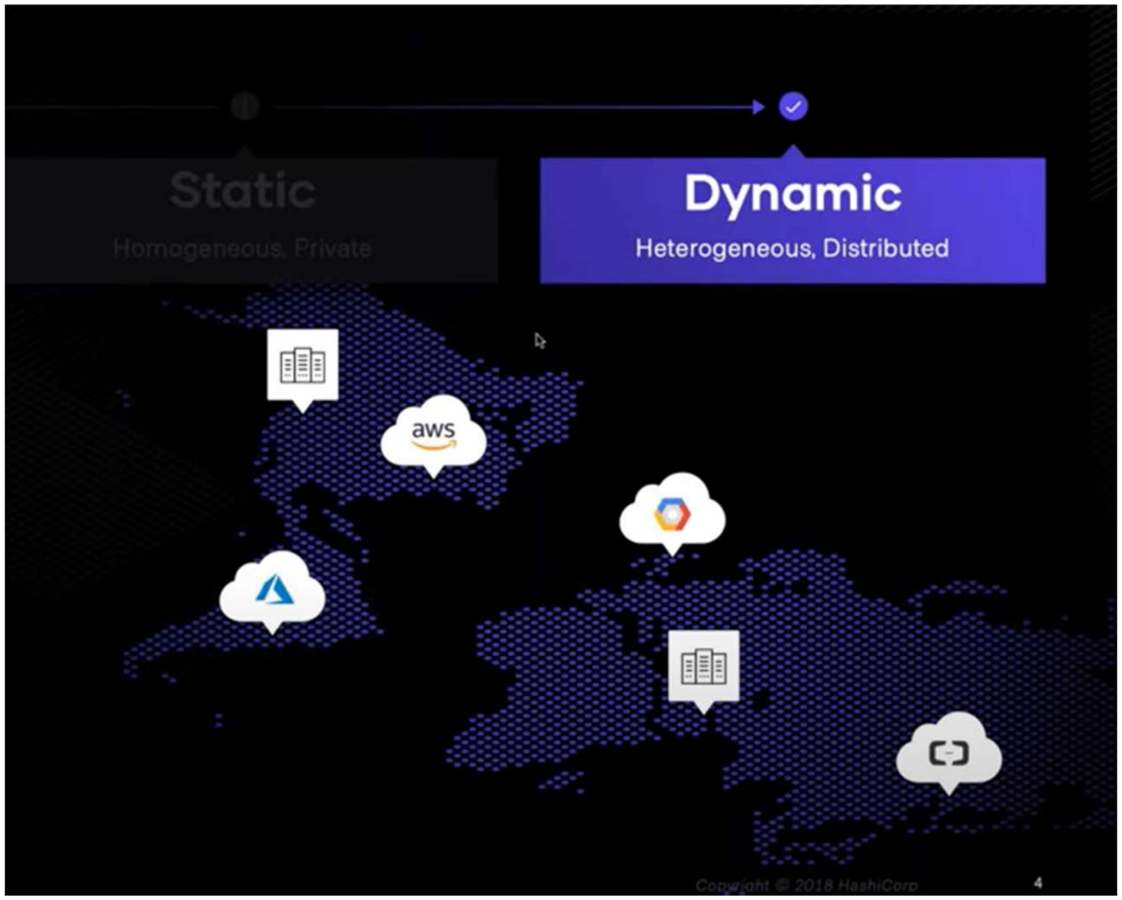
Roshan Vadagovanur Chandrasekhar
SM – Infra Dev Expert - CNAIP

Why Infrastructure as Code

- Traditional Infrastructure
  - Raise a Request very early for any infrastructure needs with operations team
  - Long waiting period
  - Difficult to make changes to the hardware once provisioned
  - Long workflows to provision new infrastructure
  - Scaling In/Out was difficult

Why Infrastructure as Code

- Dynamic Infrastructure
  - ➢Infrastructure provision automated using code
  - ➢No waiting period
  - ➢Easy to make changes in the infrastructure
  - ➢Reliable and tested infrastructure for applications to use
  - ➢Scaling In/Out can be easily achieved



What is Infrastructure as Code?

- Infrastructure as Code (IaC) is the process of managing and provisioning cloud infrastructure with machine-readable definition files.
- It allows users to easily edit and distribute configurations while ensuring the desired state of the infrastructure
- Create reproducible infrastructure configurations.
- Allows infrastructure to be easily integrated into version control mechanisms
- Provides the ability to introduce extensive automation for infrastructure management
- Eliminates the need for manual infrastructure provisioning and management.

What is Infrastructure as Code?

- Different approaches of IaC
  - An imperative approach allows users to specify the exact steps to be taken for a change, and the system does not deviate from the specified steps.
  - A declarative approach essentially means users only need to define the end requirement, and the specific tool or platform handles the steps to take in order to achieve the defined requirement.

Introduction to Terraform

History of Terraform

- Terraform was created based on the idea of having an open source, cloud-agnostic solution; a tool that could provide the same workflows, no matter what cloud you were using.
- Terraform 0.1 version was first released in July 2014
- An Enterprise version of Terraform was first released in 2018.
- Most impactful release of Terraform was 0.12

Introduction to Terraform

What is Terraform ?

HashiCorp Terraform is an infrastructure as code tool

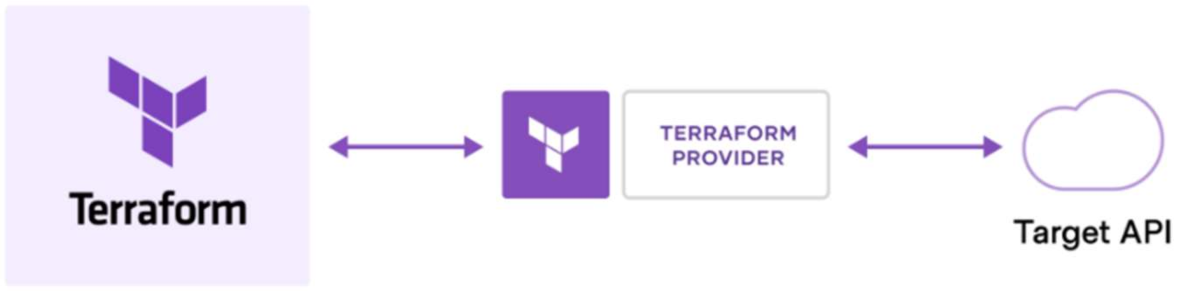
- Define both cloud and on-prem resources in human-readable configuration files
- Version the configuration files in any version control system
- Reuse the configuration files
- Share the code and collaborate with different teams
- Consistent workflow to provision and manage all of your infrastructure throughout its lifecycle.
- It ships as a single binary which is written in Go. Terraform is cross platform and can run on Linux, Windows, or MacOS.
- Installing terraform is easy. You simply download a zip file, unzip it, and run it.

Why Terraform?

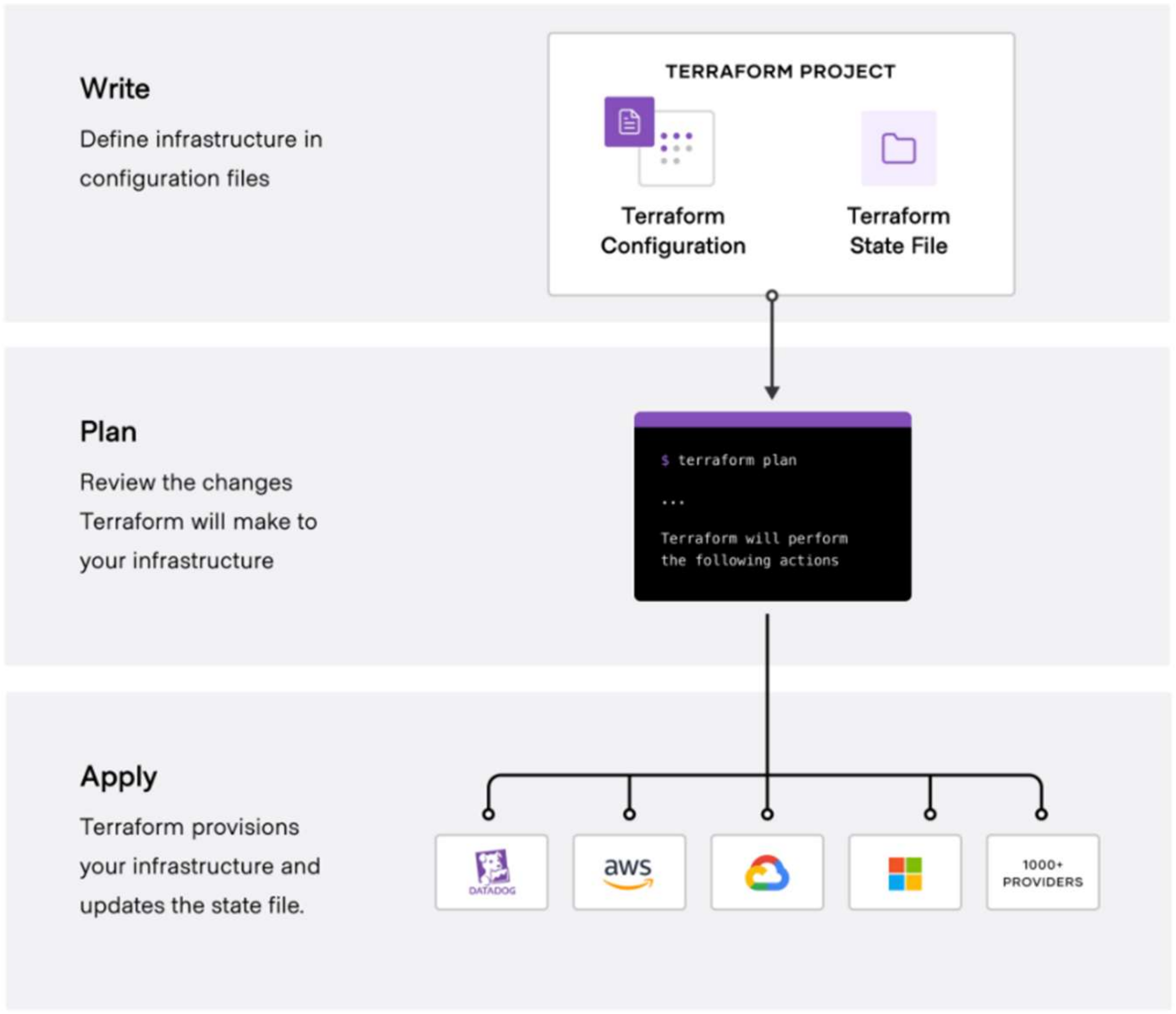
- Manage any infrastructure (multi cloud, Hybrid, On Prem)
- Track your infrastructure
- Automate changes
- Standardize configurations
- Collaborate
- Migrate from other cloud providers
- Increase provisioning speed
- Improve efficiency
- Reduce risk
- Apply incremental changes
- Destroy when needed
- Preview Changes
- Scale Easily

How Terraform Works?

- Terraform creates and manages resources on cloud platforms and other services through their application programming interfaces (APIs)
- Providers enable Terraform to work with virtually any platform or service with an accessible API.
- HashiCorp and the Terraform community have already written more than 1700 providers to manage thousands of different types of resources and services

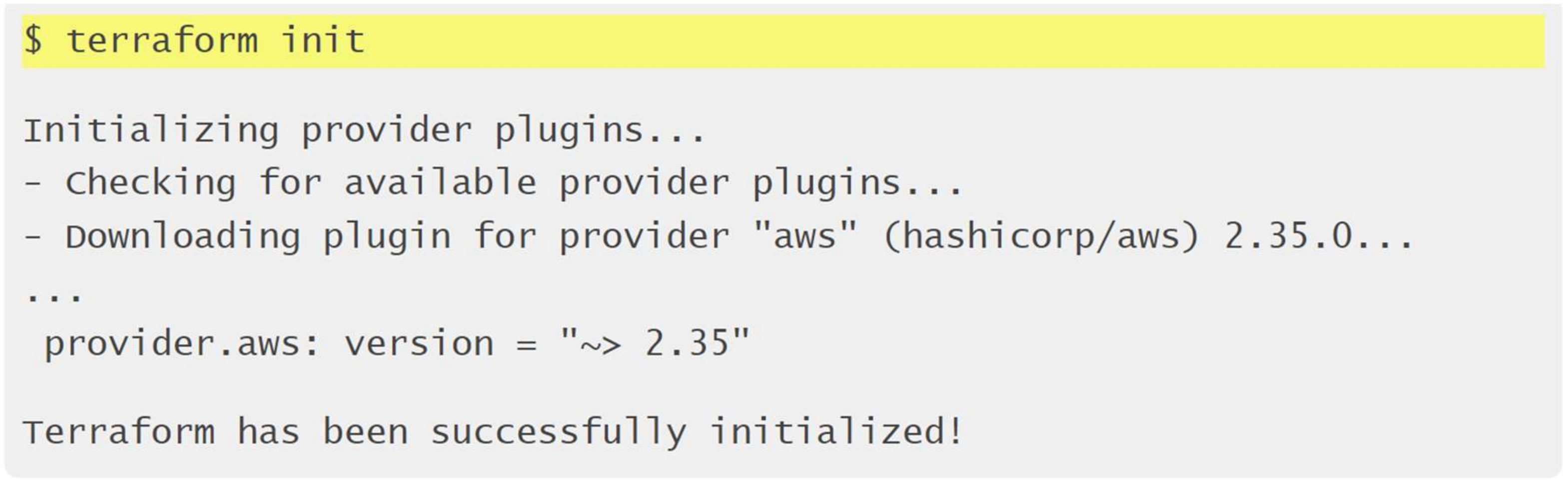


How Terraform Works?


Write
Define infrastructure in configuration files
TERRAFORM PROJECT
Terraform Configuration
Terraform State File
Plan
Review the changes Terraform will make to your infrastructure
Apply
Terraform provisions your infrastructure and updates the state file.
DATADOG
aws
1000+ PROVIDERS

```
$ terraform plan

...

Terraform will perform
the following actions
```

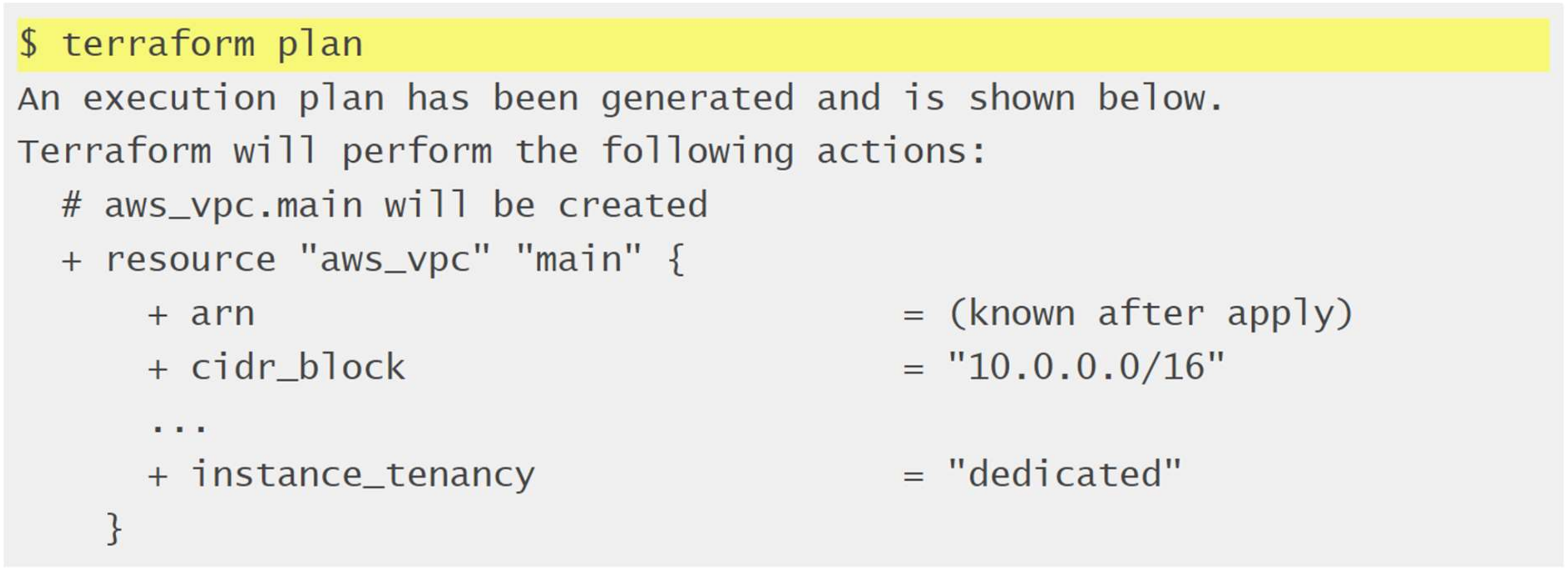
Terraform init



```
$ terraform init

Initializing provider plugins...
- Checking for available provider plugins...
- Downloading plugin for provider "aws" (hashicorp/aws) 2.35.0...
...
 provider.aws: version = "~> 2.35"

Terraform has been successfully initialized!
```

Terraform fetches any required providers and modules and stores them in the .terraform directory. If you add, change or update your modules or providers you will need to run init again

Terraform plan



```
$ terraform plan
An execution plan has been generated and is shown below.
Terraform will perform the following actions:
  # aws_vpc.main will be created
  + resource "aws_vpc" "main" {
      + arn                                  = (known after apply)
      + cidr_block                           = "10.0.0.0/16"
      ...
      + instance_tenancy                     = "dedicated"
    }
```

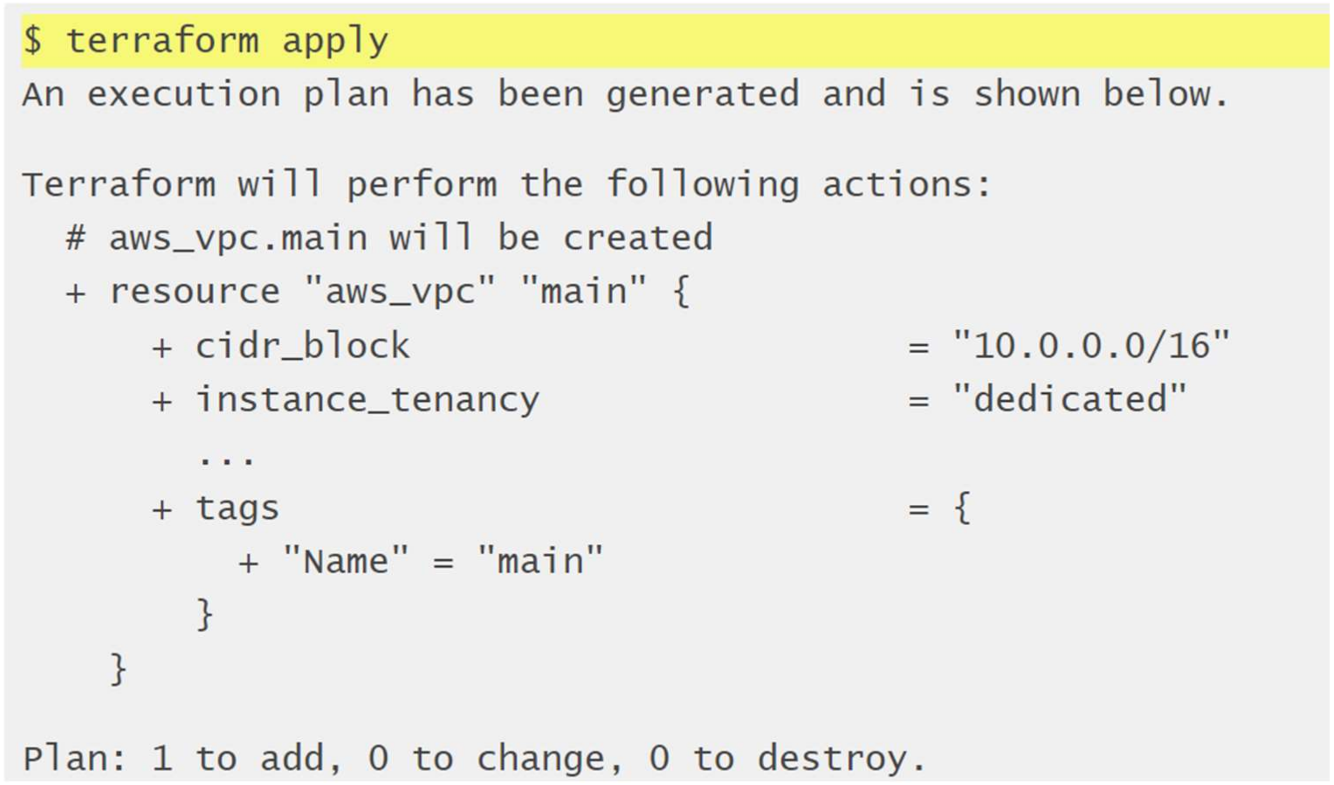
Preview your changes with terraform plan before you apply them.

 + - New changes

-   - deletions

~ - updation

Terraform Apply



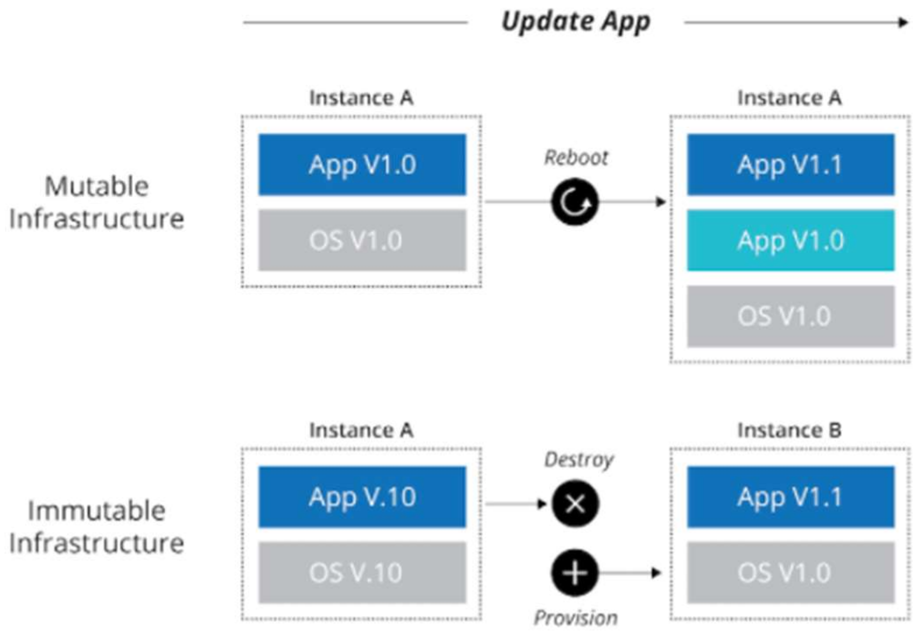
```
$ terraform apply
An execution plan has been generated and is shown below.

Terraform will perform the following actions:
  # aws_vpc.main will be created
  + resource "aws_vpc" "main" {
      + cidr_block                                = "10.0.0.0/16"
      + instance_tenancy                          = "dedicated"
        ...
      + tags                                      = {
          + "Name" = "main"
        }
    }

Plan: 1 to add, 0 to change, 0 to destroy.
```

terraform apply runs a plan and then if you approve, it applies the changes.

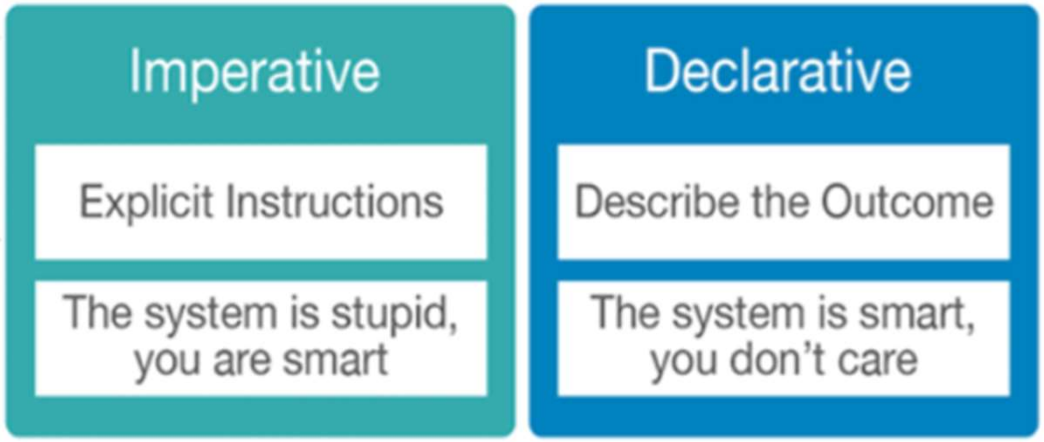
Terraform vs Alternatives

	Chef	Puppet	Ansible	CloudFormation	Terraform
Cloud		All	All	AWS	All
Type	Configuration management	Configuration management	Configuration management	Provisioning	Provisioning
Infrastructure	Mutable	Mutable	Mutable	Immutable	Immutable
Language	Procedural	Declarative	Procedural	Declarative	Declarative
Architecture	Client-Server	Client-Server	Client Only	Client only	Client only
Lifecycle management(State)	No	No	No	No	Yes
VM provisioning	Partial	Partial	Partial	Partial	Yes
Networking	Partial	Partial	Partial	Partial	Yes
Storage Management	Partial	Partial	Partial	Partial	Yes
Configuration					
Packaging	Yes	Yes	Yes	Partial	Partial
Templating	Yes	Yes	Yes	Partial	Partial

Mutable vs Immutable



Procedural vs Declarative



Imperative	Declarative
Explicit Instructions	Describe the Outcome
The system is stupid, you are smart	The system is smart, you don't care

Terraform Installation & Validation

Terraform can be downloaded from the HashiCorp Terraform website :

https://www.terraform.io/

Installation steps in RHEL/Amazon Linux/CentOS

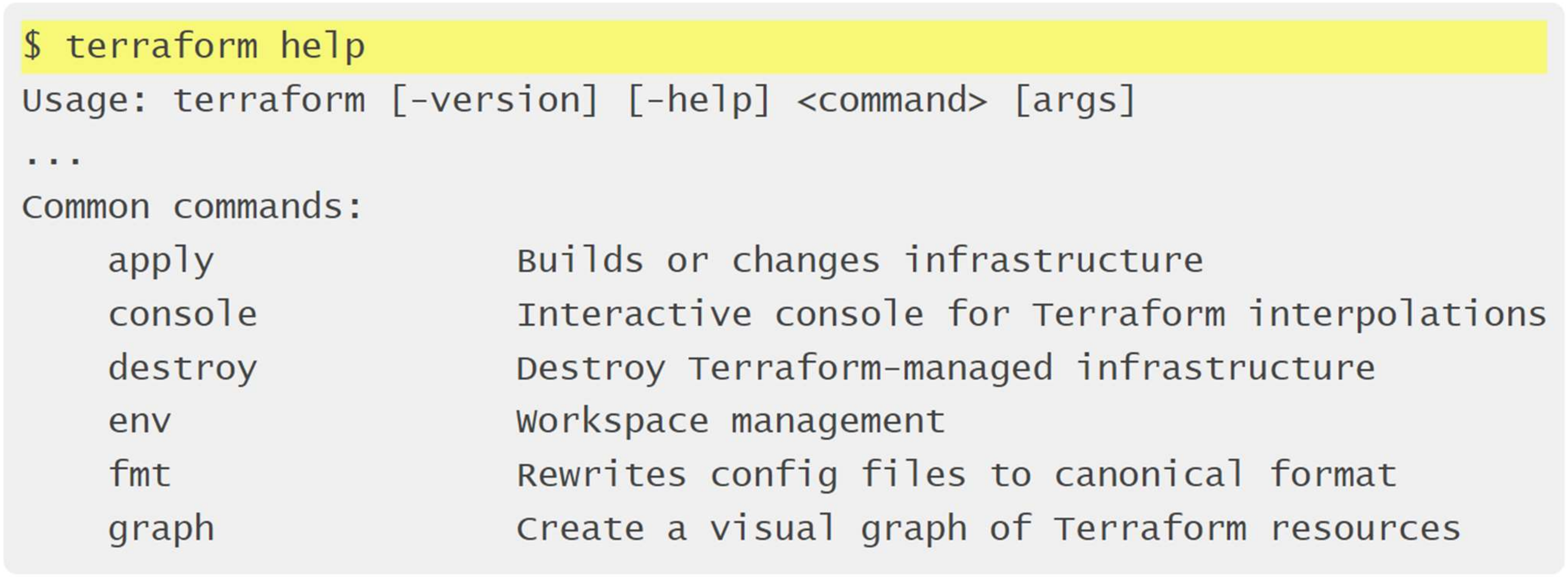
```
sudo yum install -y yum-utils
sudo yum-config-manager --add-repo https://rpm.releases.hashicorp.com/AmazonLinux/hashicorp.repo
sudo yum -y install terraform
```

Terraform Installation & Validation

Configuration steps in Windows

- *Go to Control Panel -> System -> System settings -> Environment Variables.*
- *Scroll down in system variables until you find PATH.*
- *Click edit and add the path where you have placed the Terraform Binary*
- *BE SURE to include a semicolon at the end of the previous as that is the delimiter, i.e. c:\path;c:\path2*
- *Launch a new console for the settings to take effect.*

Terraform Installation & Validation



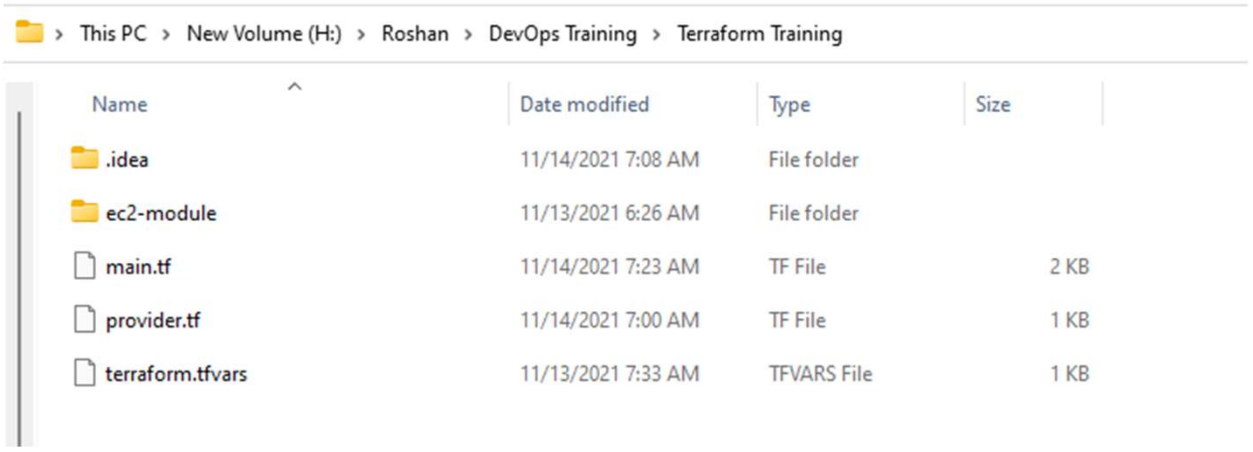
```
$ terraform help
Usage: terraform [-version] [-help] <command> [args]
...
Common commands:
    apply              Builds or changes infrastructure
    console            Interactive console for Terraform interpolations
    destroy            Destroy Terraform-managed infrastructure
    env                Workspace management
    fmt                Rewrites config files to canonical format
    graph              Create a visual graph of Terraform resources
```

Type ***terraform subcommand help*** to view help on a particular subcommand.

Terraform Language

- Files and Directories
  - File Extension

    Code in the Terraform language is stored in plain text files with the .tf file extension



This PC > New Volume (H:) > Roshan > DevOps Training > Terraform Training

Name	Date modified	Type	Size
.idea	11/14/2021 7:08 AM	File folder	
ec2-module	11/13/2021 6:26 AM	File folder	
main.tf	11/14/2021 7:23 AM	TF File	2 KB
provider.tf	11/14/2021 7:00 AM	TF File	1 KB
terraform.tfvars	11/13/2021 7:33 AM	TFVARS File	1 KB

  - Text Encoding

    Configuration files must always use UTF-8 encoding

  - Directories/Modules

    A module is a collection of .tf files kept together in a directory.

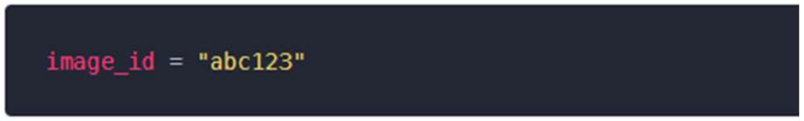
    A Terraform module only consists of the top-level configuration files in a directory; nested directories are treated as completely separate modules, and are not automatically included in the configuration.

Terraform Language

- **Syntax**
  - Configuration Syntax
    - ➢ The *native syntax* of the Terraform language, which is a rich language designed to be relatively easy for humans to read and write.
    - ➢ This low-level syntax of the Terraform language is defined in terms of a syntax called HCL

Arguments

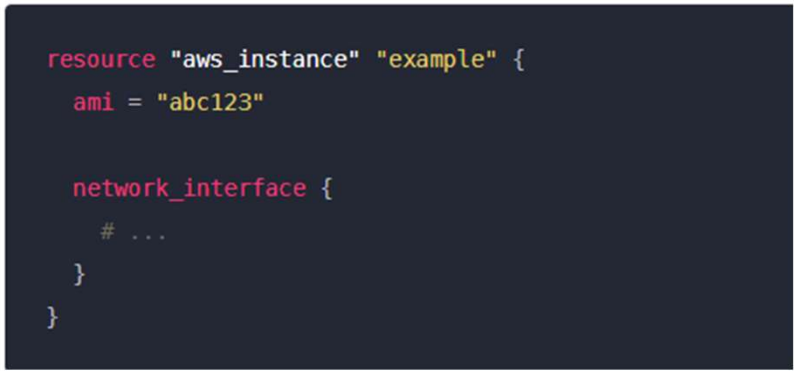
An argument assigns a value to a particular name



```
image_id = "abc123"
```

Blocks

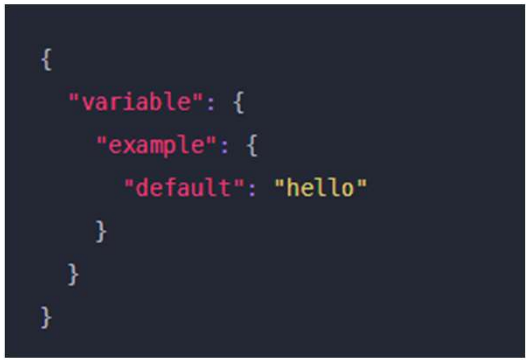
A block is a container for other content:



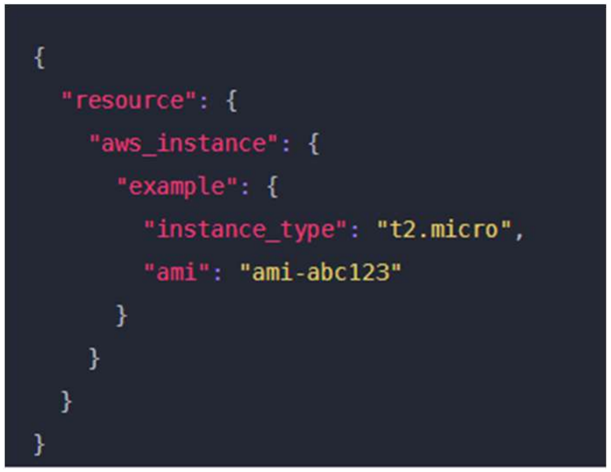
```
resource "aws_instance" "example" {
  ami = "abc123"

  network_interface {
    # ...
  }
}
```

Terraform Language

- Json Configuration Syntax
  - ➢ Terraform also supports an alternative syntax that is JSON-compatible.
  - ➢ This syntax is useful when generating portions of a configuration programmatically, since existing JSON libraries can be used to prepare the generated configuration files.
  - ➢ Terraform expects native syntax for files named with a .tf suffix, and JSON syntax for files named with a .tf.json suffix.



```
{
  "variable": {
    "example": {
      "default": "hello"
    }
  }
}
```



```
{
  "resource": {
    "aws_instance": {
      "example": {
        "instance_type": "t2.micro",
        "ami": "ami-abc123"
      }
    }
  }
}
```

Terraform Providers

Terraform relies on plugins called "providers" to interact with remote systems. Terraform configurations must declare which providers they require, so that Terraform can install and use them.

- What Providers Do
  - Each provider adds a set of resource types and/or data sources that Terraform can manage.
  - Every resource type is implemented by a provider; without providers, Terraform can't manage any kind of infrastructure.
- Where Providers Come From
  - Providers are distributed separately from Terraform itself, and each provider has its own release cadence and version numbers.

Terraform Providers

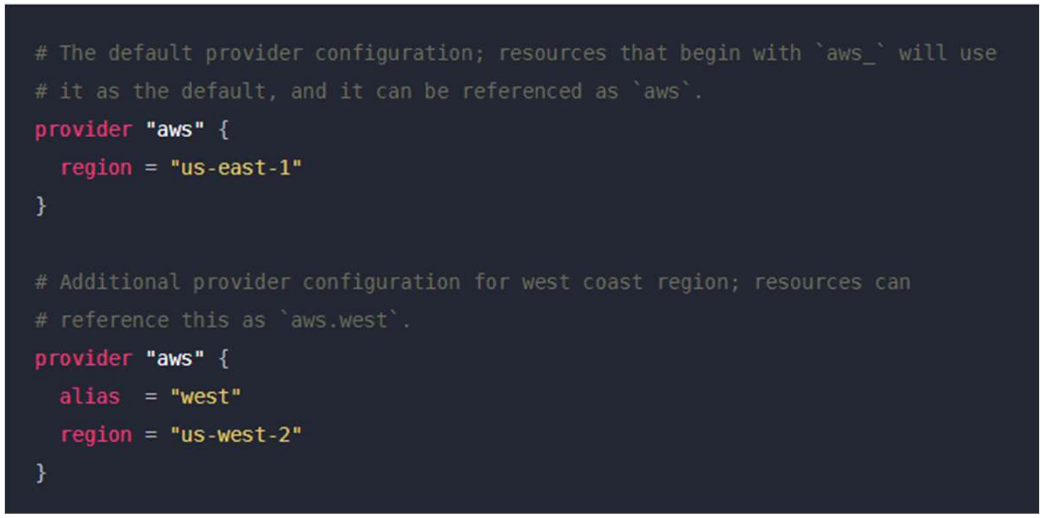
- Provider Configuration
  - Providers allow Terraform to interact with cloud providers, SaaS providers, and other APIs.
  - Provider configurations belong in the root module of a Terraform configuration.
  - Every resource type is implemented by a provider; without providers, Terraform can't manage any kind of infrastructure.

```
provider "aws" {
  region     = "us-east-1"
  access_key = “A2wdf033”
  secret_key = “032ksdf932332434”
}
```

Terraform Providers

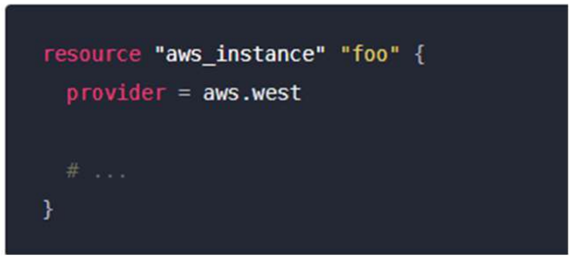
- alias: Multiple Provider Configurations

A provider block without an alias argument is the default configuration for that provider



```
# The default provider configuration; resources that begin with `aws_` will use
# it as the default, and it can be referenced as `aws`.
provider "aws" {
  region = "us-east-1"
}

# Additional provider configuration for west coast region; resources can
# reference this as `aws.west`.
provider "aws" {
  alias  = "west"
  region = "us-west-2"
}
```

To use an alternate provider configuration for a resource or data source, set its provider meta-argument to a <PROVIDER NAME>.<ALIAS> reference:

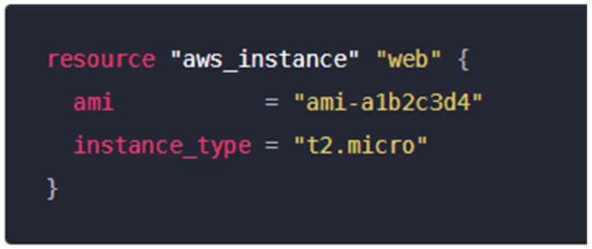


```
resource "aws_instance" "foo" {
  provider = aws.west

  # ...
}
```

Terraform Resources

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

Resource Syntax



```
resource "aws_instance" "web" {
  ami           = "ami-a1b2c3d4"
  instance_type = "t2.micro"
}
```

Resource Types

- Each resource is associated with a single resource type, which determines the kind of infrastructure object it manages and what arguments and other attributes the resource supports.

Terraform Resources

Anatomy of a Resource

```
resource type "name" {
  parameter = "foo"
  parameter2 = "bar"
  list = ["one", "two", "three"]
}
```

Every terraform resource is structured exactly the same way

resource = Top level keyword
type = Type of resource. Example: aws_instance.
name = Arbitrary name to refer to this resource. Used internally by terraform. This field cannot be a variable.

Terraform Resources

Resource Behavior

- *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.
- *Destroy and re-create* resources whose arguments have changed but which cannot be updated in-place due to remote API limitations.

Resource Dependencies

- Most resources in a configuration don't have any particular relationship, and Terraform can make changes to several unrelated resources in parallel.
- Resource dependencies are handled automatically by Terraform

Terraform Resources

Meta-Arguments

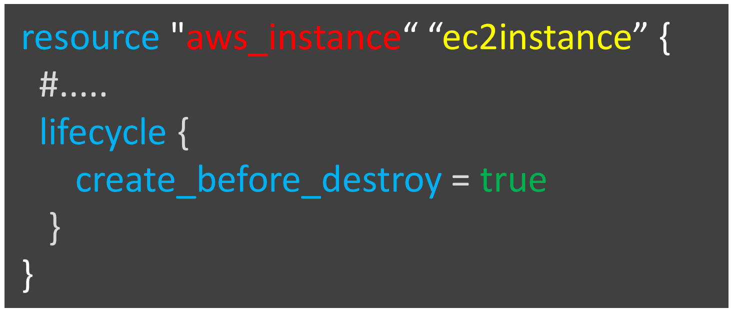
- *Terraform language defines several meta-arguments, which can be used with any resource type to change the behavior of resources.*

  - *depends_on*
  - *count*
  - *for_each*
  - *provider*
  - *lifecycle*
  - *provisioner*

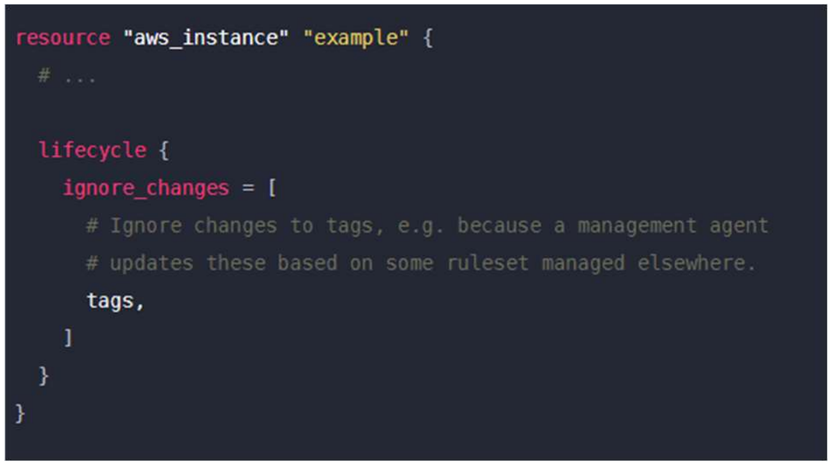
Terraform Resources

Meta-Argument: lifecycle

- *Syntax and Arguments*
  - *create_before_destroy – boolean*
  - *prevent_destroy – boolean*
  - *ignore_changes - list*



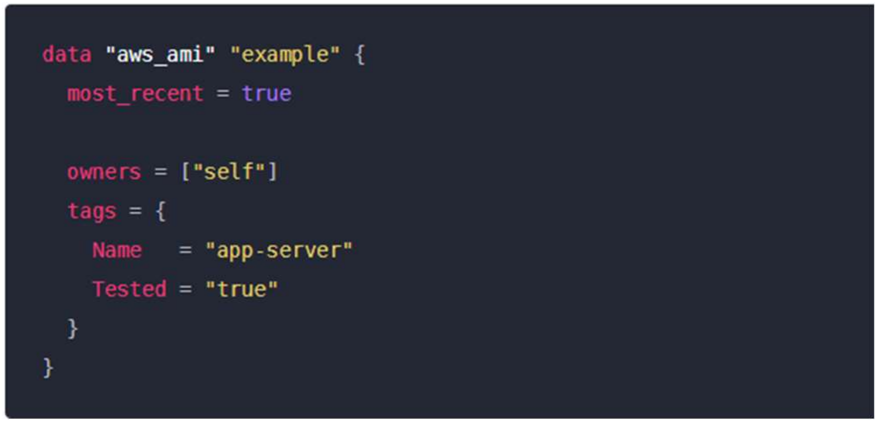
```
resource "aws_instance" "ec2instance" {
 #.....
 lifecycle {
    create_before_destroy = true
  }
}
```



```
resource "aws_instance" "example" {
  # ...

  lifecycle {
    ignore_changes = [
      # Ignore changes to tags, e.g. because a management agent
      # updates these based on some ruleset managed elsewhere.
      tags,
    ]
  }
}
```

Terraform DataSources

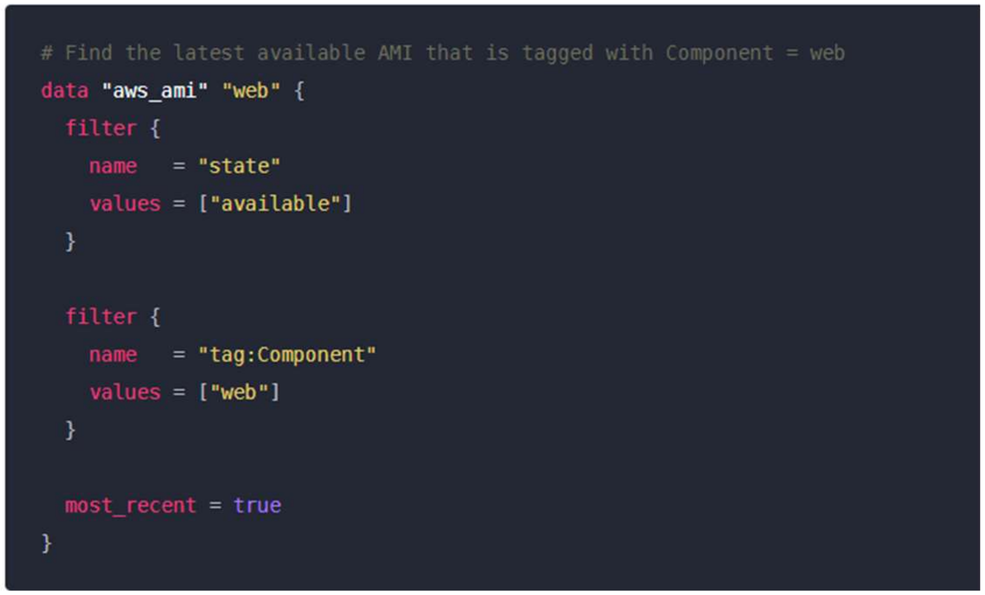
- Data sources allow Terraform to use information defined outside of Terraform, defined by another separate Terraform configuration, or modified by functions.
- A data source is accessed via a special kind of resource known as a data resource, declared using a data block
- If the query constraint arguments for a data resource refer only to constant values or values that are already known, the data resource will be read prior to creating a plan.
- Query constraint arguments may refer to values that cannot be determined until after configuration is applied. In this case, reading from the data source is deferred until the apply phase



```
data "aws_ami" "example" {
  most_recent = true

  owners = ["self"]
  tags = {
    Name   = "app-server"
    Tested = "true"
  }
}
```

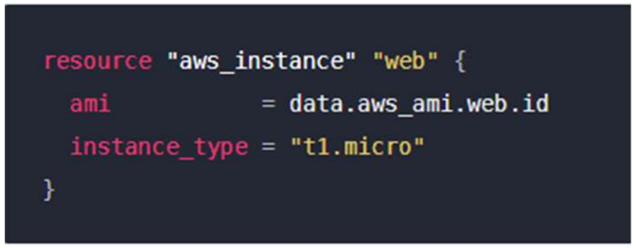
Terraform DataSources

- Example



```
# Find the latest available AMI that is tagged with Component = web
data "aws_ami" "web" {
  filter {
    name   = "state"
    values = ["available"]
  }

  filter {
    name   = "tag:Component"
    values = ["web"]
  }

  most_recent = true
}
```

- Each data instance will export one or more attributes, which can be used in other resources as reference expressions of the form data.<TYPE>.<NAME>.<ATTRIBUTE>.

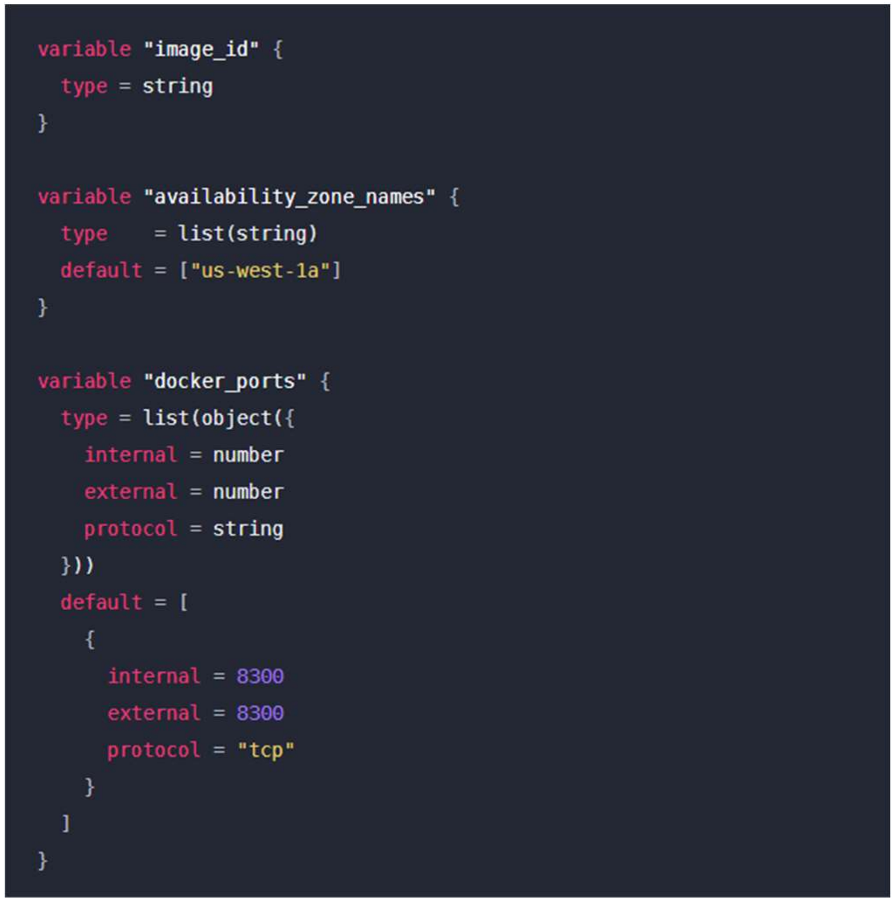


```
resource "aws_instance" "web" {
  ami           = data.aws_ami.web.id
  instance_type = "t1.micro"
}
```

Terraform Variables

Declaring an Input Variable

- Each input variable accepted by a module must be declared using a variable block
- The label after the *variable* keyword is a name for the variable, which must be unique among all variables
- The name of a variable can be any valid identifier except the following: source, version, providers, count, for_each, lifecycle, depends_on, locals.

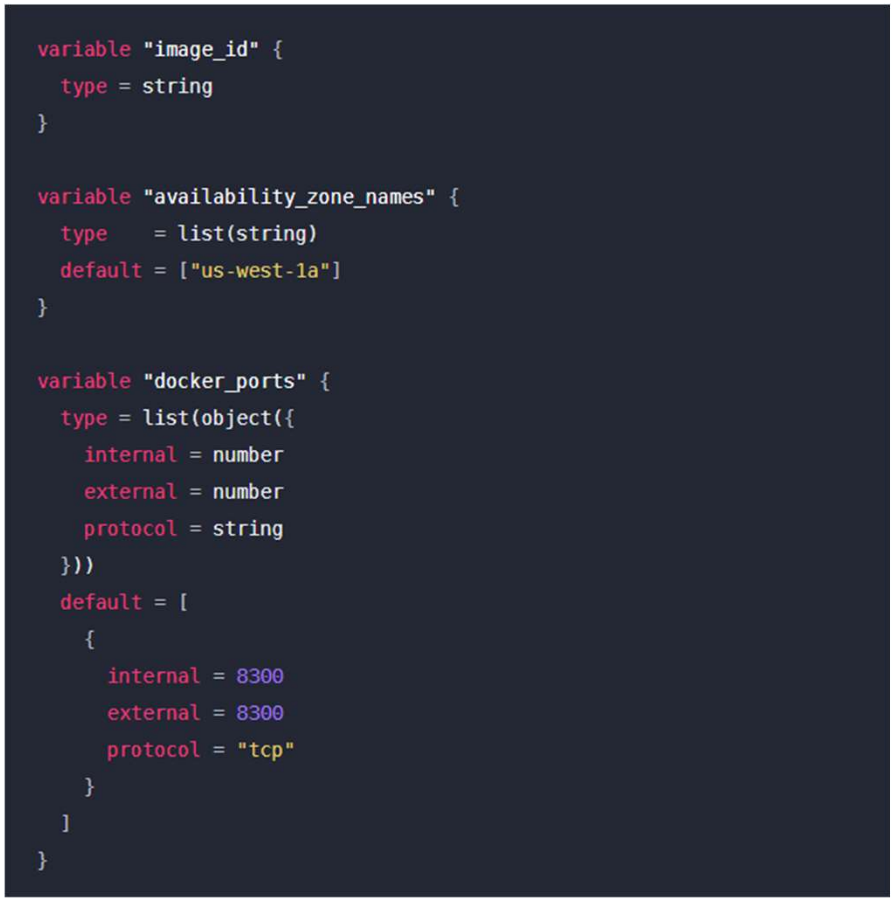


```
variable "image_id" {
  type = string
}

variable "availability_zone_names" {
  type    = list(string)
  default = ["us-west-1a"]
}

variable "docker_ports" {
  type = list(object({
    internal = number
    external = number
    protocol = string
  }))
  default = [
    {
      internal = 8300
      external = 8300
      protocol = "tcp"
    }
  ]
}
```

Terraform Variables

Arguments

- *default* - A default value which then makes the variable optional.
- *type* - This argument specifies what value types are accepted for the variable.
- *description* - This specifies the input variable's documentation.
- *validation* - A block to define validation rules, usually in addition to type constraints.
- *sensitive* - Limits Terraform UI output when the variable is used in configuration.
- *nullable* - Specify if the variable can be null within the module.



```
variable "image_id" {
  type = string
}

variable "availability_zone_names" {
  type    = list(string)
  default = ["us-west-1a"]
}

variable "docker_ports" {
  type = list(object({
    internal = number
    external = number
    protocol = string
  }))
  default = [
    {
      internal = 8300
      external = 8300
      protocol = "tcp"
    }
  ]
}
```

Terraform Variables

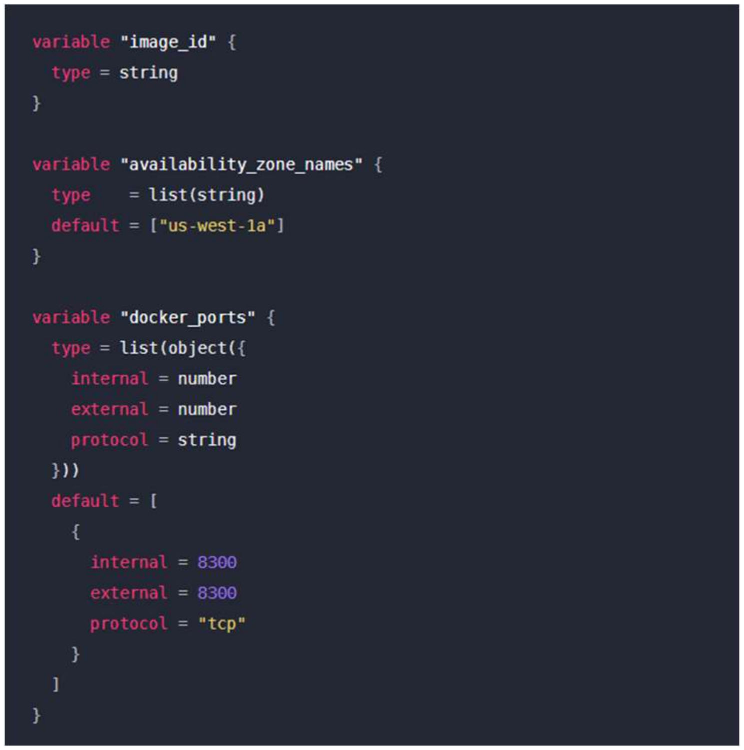
Type constraints

- The type argument in a variable block allows you to restrict the type of value that will be accepted as the value for a variable.
- If no type constraint is set then a value of any type is accepted.
- Type constraints are created from a mixture of type keywords and type constructors. The supported type keywords are:

string a sequence of Unicode characters representing some text, like "hello"

number a numeric value. The number type can represent both whole numbers like 15 and fractional values like 6.283185.

bool a boolean value, either true or false



```
variable "image_id" {
  type = string
}

variable "availability_zone_names" {
  type    = list(string)
  default = ["us-west-1a"]
}

variable "docker_ports" {
  type = list(object({
    internal = number
    external = number
    protocol = string
  }))
  default = [
    {
      internal = 8300
      external = 8300
      protocol = "tcp"
    }
  ]
}
```

Terraform Variables

Type constraints

The type constructors allow you to specify complex types such as collections:

list(<TYPE>) a sequence of values, like ["us-west-1a", "us-west-1c"]

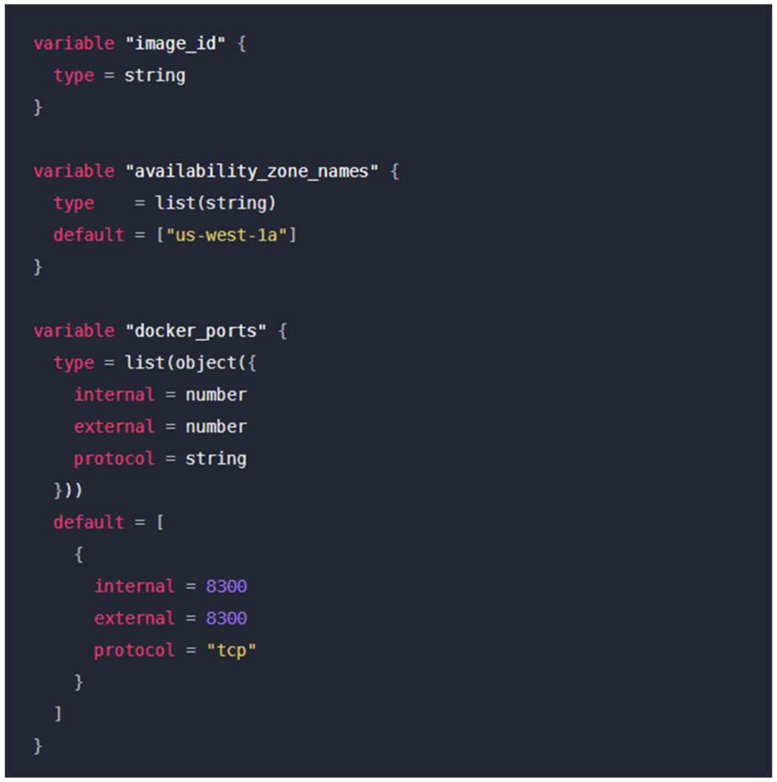
set(<TYPE>) a sequence of values

map(<TYPE>) a group of values identified by named labels, like {name = "Mabel", age = 52}.

object({<ATTR NAME> = <TYPE>, ... }) a group of values identified by named labels, like {name = "Mabel", age = 52}.

tuple([<TYPE>, ...]) a sequence of values, like ["us-west-1a", "us-west-1c"]

any

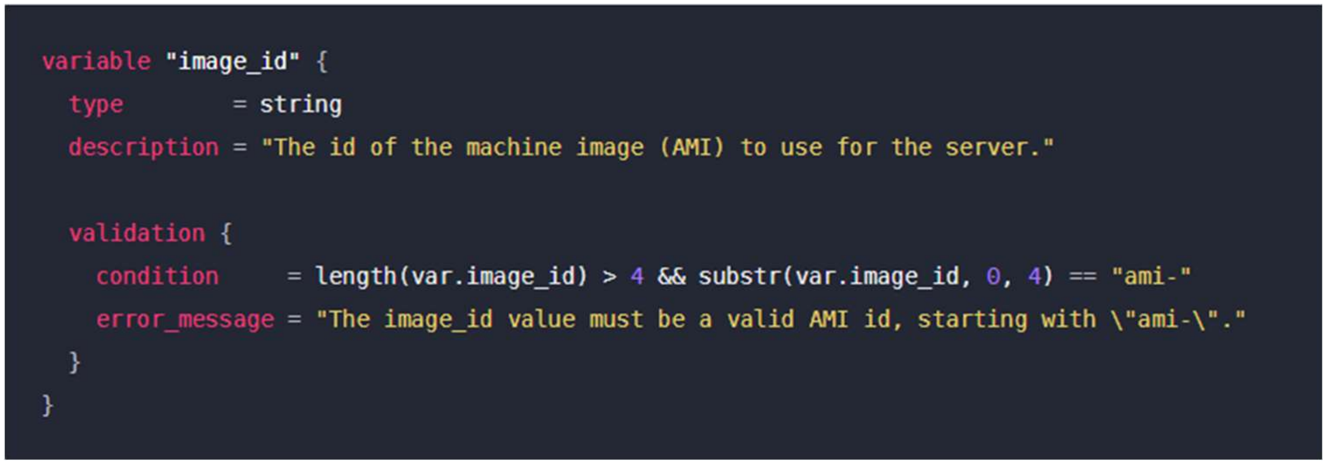


```
variable "image_id" {
  type = string
}

variable "availability_zone_names" {
  type    = list(string)
  default = ["us-west-1a"]
}

variable "docker_ports" {
  type = list(object({
    internal = number
    external = number
    protocol = string
  }))
  default = [
    {
      internal = 8300
      external = 8300
      protocol = "tcp"
    }
  ]
}
```

Terraform Variables

Variable Value Validation

- In addition to Type Constraints, custom validation rules for a particular variable can be specified using a *validation* block nested within the corresponding variable block
- The condition argument is an expression that must use the value of the variable to return true

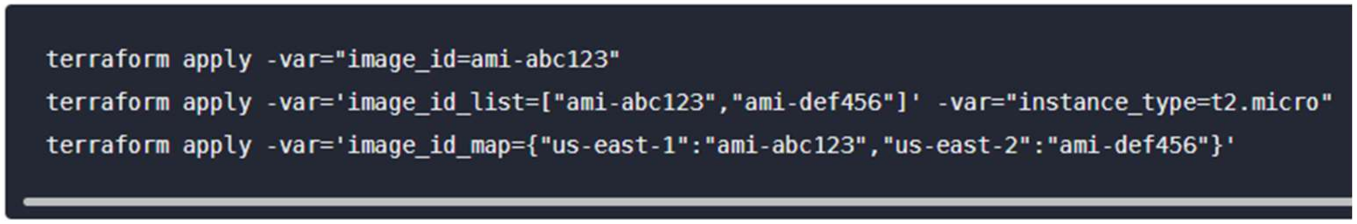


```
variable "image_id" {
  type        = string
  description = "The id of the machine image (AMI) to use for the server."

  validation {
    condition     = length(var.image_id) > 4 && substr(var.image_id, 0, 4) == "ami-"
    error_message = "The image_id value must be a valid AMI id, starting with \"ami-\"."
  }
}
```

Terraform Variables

Variables on the Command Line

- To specify individual variables on the command line, use the -var option when running the terraform plan and terraform apply commands:

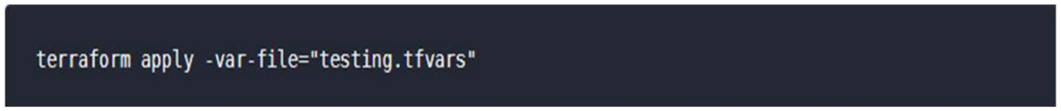


```
terraform apply -var="image_id=ami-abc123"
terraform apply -var='image_id_list=["ami-abc123","ami-def456"]' -var="instance_type=t2.micro"
terraform apply -var='image_id_map={"us-east-1":"ami-abc123","us-east-2":"ami-def456"}'
```

Terraform Variables

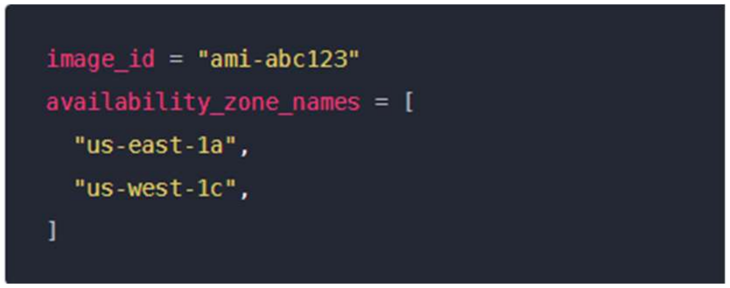
Variable Definitions (.tfvars) Files

- To set lots of variables, it is more convenient to specify their values in a variable definitions file (*with a filename ending in either .tfvars or .tfvars.json*) and then specify that file on the command line with -var-file:



```
terraform apply -var-file="testing.tfvars"
```

- A variable definitions file uses the same basic syntax as Terraform language files, but consists only of variable name assignments

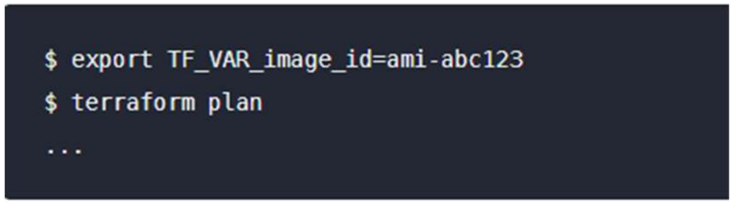


```
image_id = "ami-abc123"
availability_zone_names = [
  "us-east-1a",
  "us-west-1c",
]
```

- 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*.

Environment Variables

- Terraform searches the environment of its own process for environment variables named TF_VAR_ followed by the name of a declared variable.



```
$ export TF_VAR_image_id=ami-abc123
$ terraform plan
...
```

Terraform Variables

Variable precedence

Terraform loads variables in the following order, with later sources taking precedence over earlier ones:

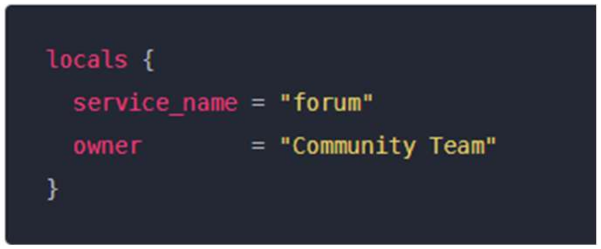
- Environment variables
- The *terraform.tfvars* file, if present.
- The *terraform.tfvars.json* file, if present.
- Any **.auto.tfvars* or **.auto.tfvars.json* files, processed in lexical order of their filenames.
- Any *-var* and *-var-file* options on the command line, in the order they are provided.

Terraform Variables

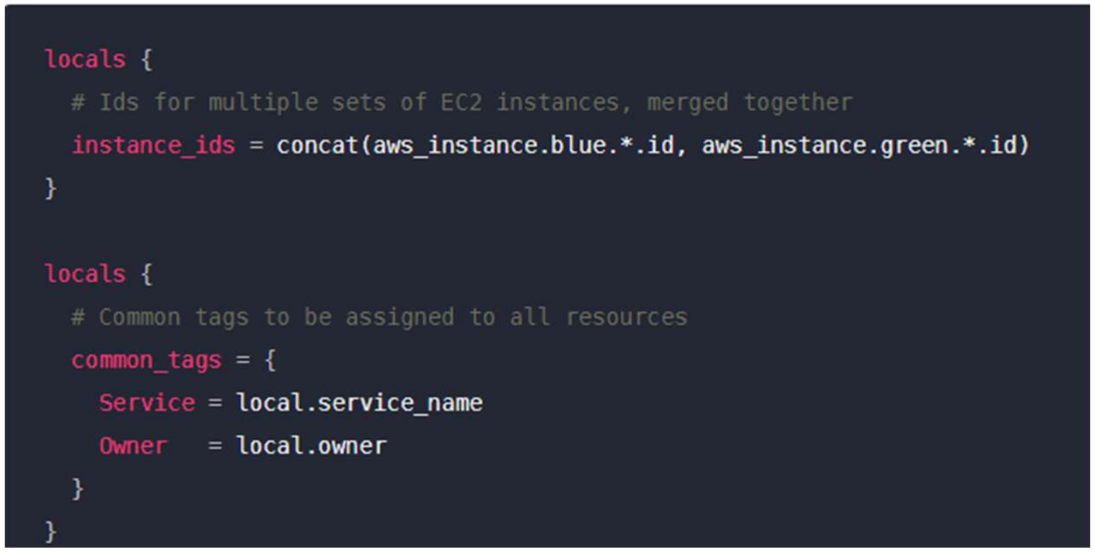
Local Values

Declaring a Local Value

- A set of related local values can be declared together in a single locals block:



```
locals {
  service_name = "forum"
  owner        = "Community Team"
}
```



```
locals {
  # Ids for multiple sets of EC2 instances, merged together
  instance_ids = concat(aws_instance.blue.*.id, aws_instance.green.*.id)
}

locals {
  # Common tags to be assigned to all resources
  common_tags = {
    Service = local.service_name
    Owner   = local.owner
  }
}
```

Local Values

Using Local Values

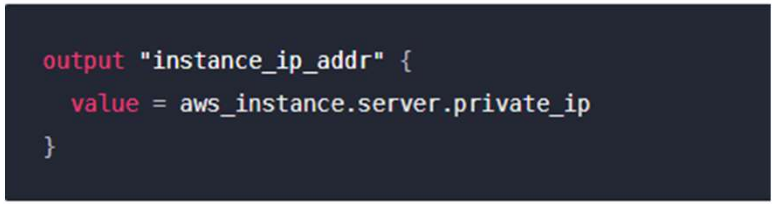
- Once a local value is declared, you can reference it as local.<NAME>.

Terraform Output

Output values are similar to return values in programming languages.

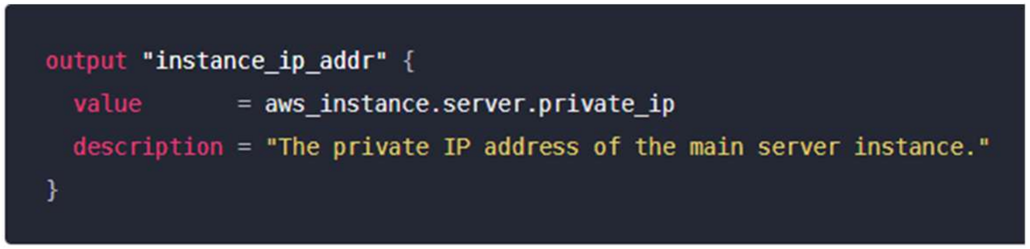
Declaring an Output Value

Each output value exported by a module must be declared using an *output* block



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

output blocks can optionally include *description*, *sensitive*, and *depends_on* arguments, which are described in the following sections.

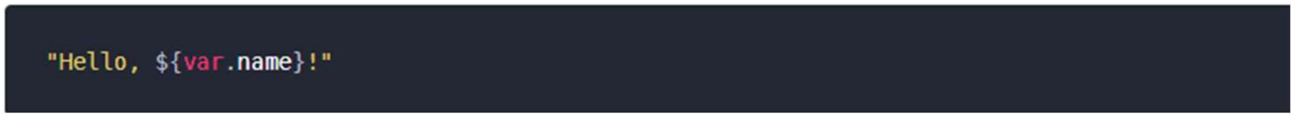


```
output "instance_ip_addr" {
  value       = aws_instance.server.private_ip
  description = "The private IP address of the main server instance."
}
```

Terraform String Templates

- **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



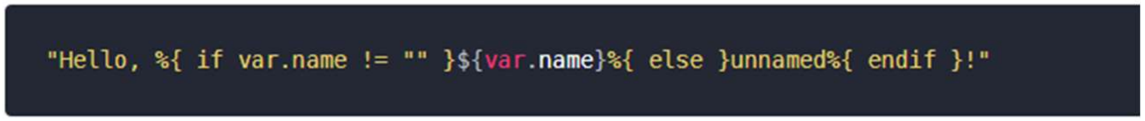
```
"Hello, ${var.name}!"
```

Terraform String Templates

Directives

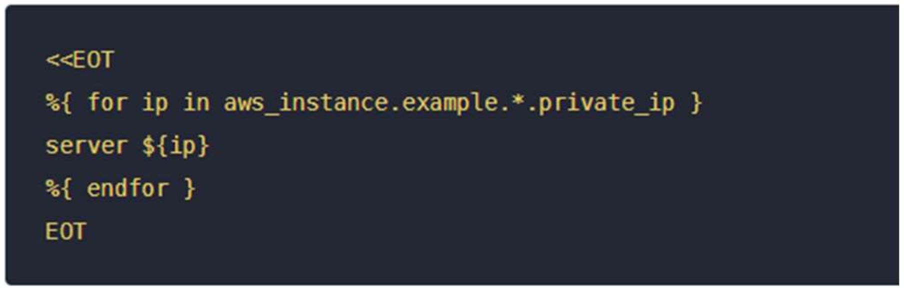
A %{ ... } sequence is a directive, which allows for conditional results and iteration over collections, similar to conditional and for expressions.

The following directives are supported:



```
"Hello, %{ if var.name != "" }${var.name}%{ else }unnamed%{ endif }!"
```

- The %{if <BOOL>}/%{else}/%{endif} directive chooses between two templates based on the value of a bool expression:



```
<<EOT
%{ for ip in aws_instance.example.*.private_ip }
server ${ip}
%{ endfor }
EOT
```

- The %{for <NAME> in <COLLECTION>} / %{endfor} directive iterates over the elements of a given collection or structural value and evaluates a given template once for each element, concatenating the results together

Terraform Operators

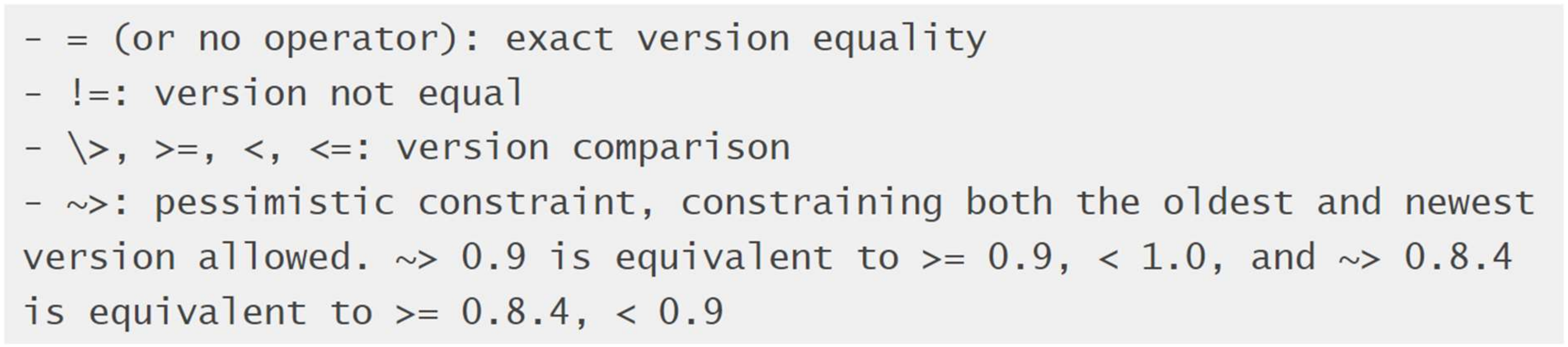
Arithmetic Operators : +, -, *, /, %

Equality Operators: ==, !=

Comparison Operators : < , <=, >,>=

Logical Operators: ||, &&, !

Versioning Operators :

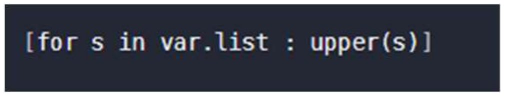


```
- = (or no operator): exact version equality
- !=: version not equal
- \>, >=, <, <=: version comparison
- ~>: pessimistic constraint, constraining both the oldest and newest
version allowed. ~> 0.9 is equivalent to >= 0.9, < 1.0, and ~> 0.8.4
is equivalent to >= 0.8.4, < 0.9
```

Terraform For Expression

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

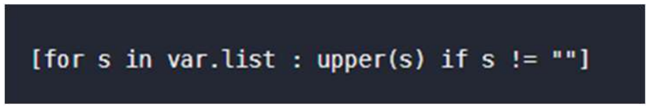
Each element in the input value can correspond to either one or zero values in the result



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

Filtering Elements

A *for* expression can also include an optional *if* clause to filter elements from the source collection, producing a value with fewer elements than the source value

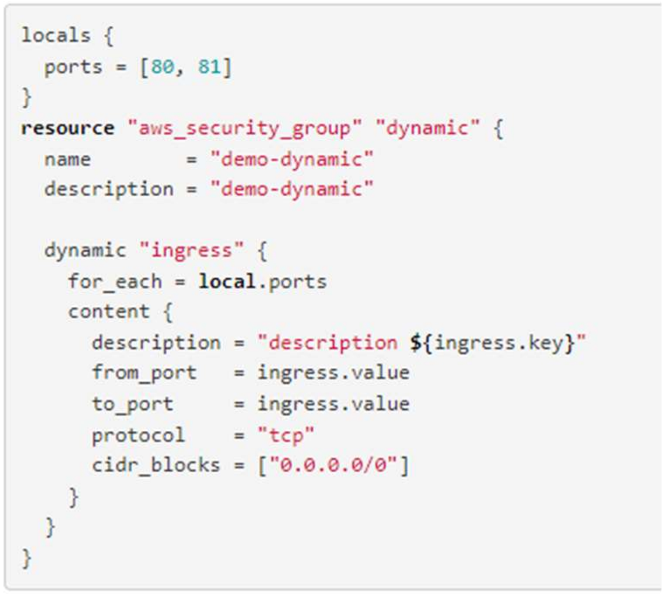


```
[for s in var.list : upper(s) if s != ""]
```

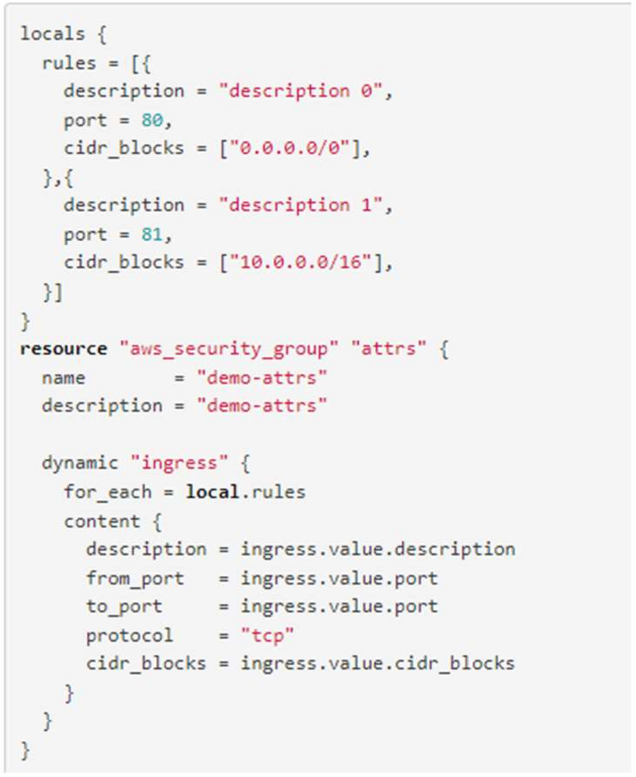
Terraform Dynamic blocks

A *dynamic* block acts much like a for expression, but 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.



```
locals {
  ports = [80, 81]
}
resource "aws_security_group" "dynamic" {
  name        = "demo-dynamic"
  description = "demo-dynamic"

  dynamic "ingress" {
    for_each = local.ports
    content {
      description = "description ${ingress.key}"
      from_port   = ingress.value
      to_port     = ingress.value
      protocol    = "tcp"
      cidr_blocks = ["0.0.0.0/0"]
    }
  }
}
```



```
locals {
  rules = [{
    description = "description 0",
    port = 80,
    cidr_blocks = ["0.0.0.0/0"],
  },{
    description = "description 1",
    port = 81,
    cidr_blocks = ["10.0.0.0/16"],
  }]
}
resource "aws_security_group" "attrs" {
  name        = "demo-attrs"
  description = "demo-attrs"

  dynamic "ingress" {
    for_each = local.rules
    content {
      description = ingress.value.description
      from_port   = ingress.value.port
      to_port     = ingress.value.port
      protocol    = "tcp"
      cidr_blocks = ingress.value.cidr_blocks
    }
  }
}
```

Terraform count and for_each

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.

count is sensible for any changes in list order, this means that if for some reason order of the list is changed, terraform will force replacement of all resources of which the index in the list has changed

```
variable "my_list" {
  default = ["first", "second", "third"]
}

resource "null_resource" "default" {
  count = length(var.my_list)
  triggers = {
    list_index = count.index
    list_value = var.my_list[count.index]
  }
}
```

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.

for_each helps to avoid accidental / unwanted recreation of resource instances when a input list (or just it’s order) has been modified.

```
variable "my_list" {
  default = ["zero", "first", "second", "third"]
}

resource "null_resource" "default" {
  for_each = toset(var.my_list)
  triggers = {
    list_index = each.key
    list_value = each.value
  }
}
```

Terraform Built in Functions

- The Terraform language includes a number of built-in functions that you can call from within expressions to transform and combine values.
- The general syntax for function calls is a function name followed by comma-separated arguments in parentheses:

  max(5, 12, 9)

- The Terraform language does not support user-defined functions, and so only the functions built in to the language are available for use.
- Available functions
  - Numeric Functions : min(), max(),ceil(), floor()
  - String functions: chomp(),join(),substr(),split(),lower(),upper()
  - Collection functions : concat(), distinct(),length(),lookup()
  - Filesystem functions: dirname(),basename(),file(),fileexists(),templatefile()
  - Date and Time functions: timestamp()
  - Type conversion functions : toset(),tolist(),tostring(),tonumber(), tomap()

Terraform modules

What are modules for?

- Organize configuration: Modules make it easier to navigate, understand, and update your configuration by keeping related parts of your configuration together

- Encapsulate configuration - Another benefit of using modules is to encapsulate configuration into distinct logical components

- Re-use configuration - Writing all of your configuration from scratch can be time consuming and error prone.

- Provide consistency and ensure best practices - Modules also help to provide consistency in your configurations.

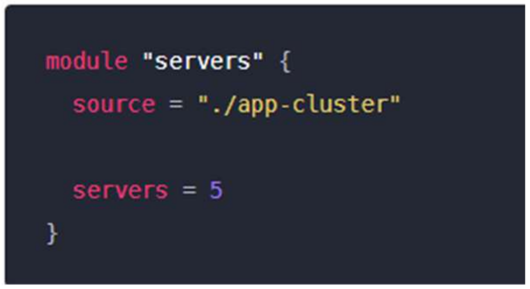
Terraform modules

- Modules are containers for multiple resources that are used together.
- A module consists of a collection of .tf and/or .tf.json files kept together in a directory.
- Modules are the main way to package and reuse resource configurations with Terraform.
- **The Root Module**: Every Terraform configuration has at least one module, known as its root module, which consists of the resources defined in the .tf files in the main working directory
- **Child Modules** : A Terraform module (usually the root module of a configuration) can call other modules to include their resources into the configuration. A module that has been called by another module is often referred to as a child module.

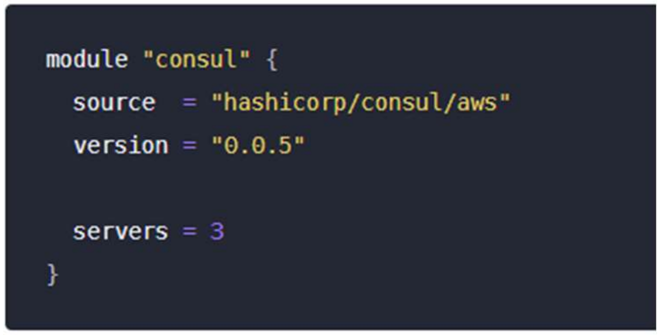
Terraform modules

Calling a Child Module

- 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,
- 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 defined by the module.
  - Terraform defines a few other meta-arguments that can be used with all modules, including for_each and depends_on



```
module "servers" {
  source = "./app-cluster"

  servers = 5
}
```

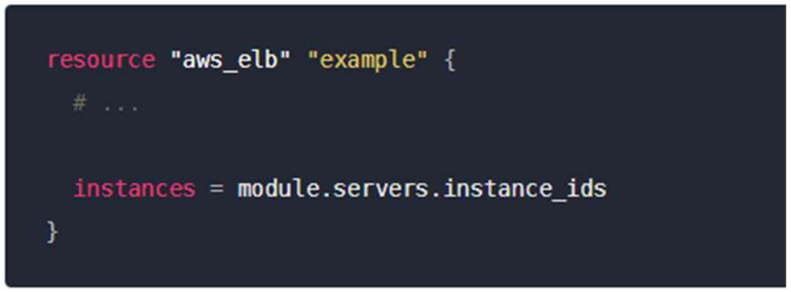


```
module "consul" {
  source  = "hashicorp/consul/aws"
  version = "0.0.5"

  servers = 3
}
```

Terraform modules

Accessing Module Output Values

- The resources defined in a module are encapsulated, so the calling module cannot access their attributes directly. However, the child module can declare output values to selectively export certain values to be accessed by the calling module.

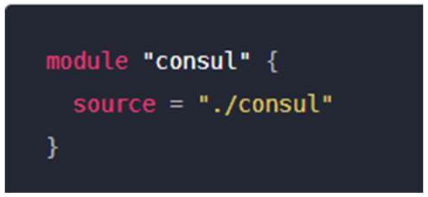


```
resource "aws_elb" "example" {
  # ...

  instances = module.servers.instance_ids
}
```

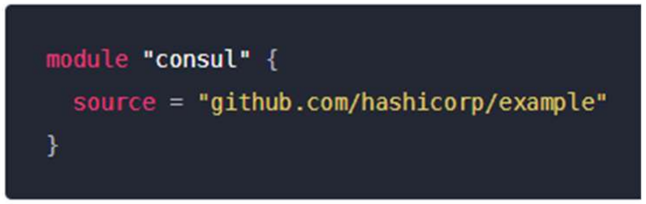
Terraform modules

Module Sources

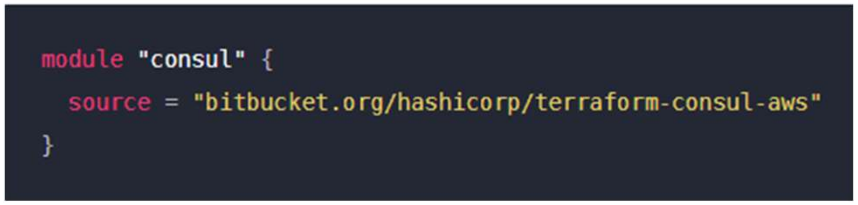
- The module installer supports installation from a number of different source types, as listed below.
- Local paths
- Terraform Registry
- GitHub
- Bitbucket
- Generic Git, Mercurial repositories
- HTTP URLs
- S3 buckets
- GCS buckets
- Modules in Package Sub-directories



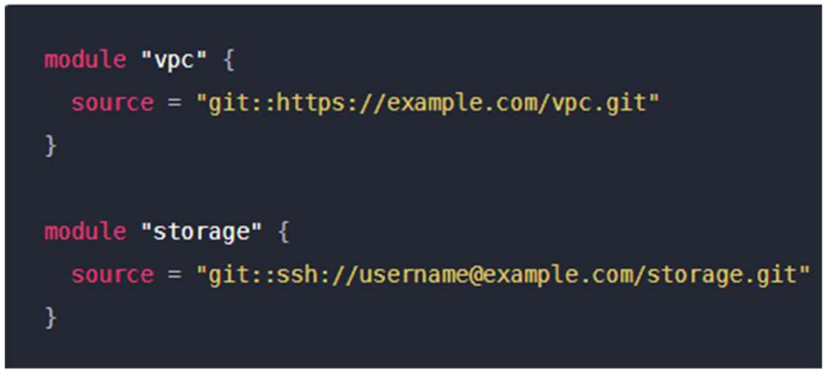
```
module "consul" {
  source = "./consul"
}
```



```
module "consul" {
  source = "github.com/hashicorp/example"
}
```



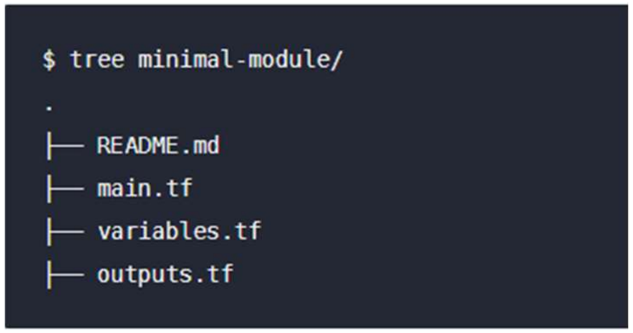
```
module "consul" {
  source = "bitbucket.org/hashicorp/terraform-consul-aws"
}
```



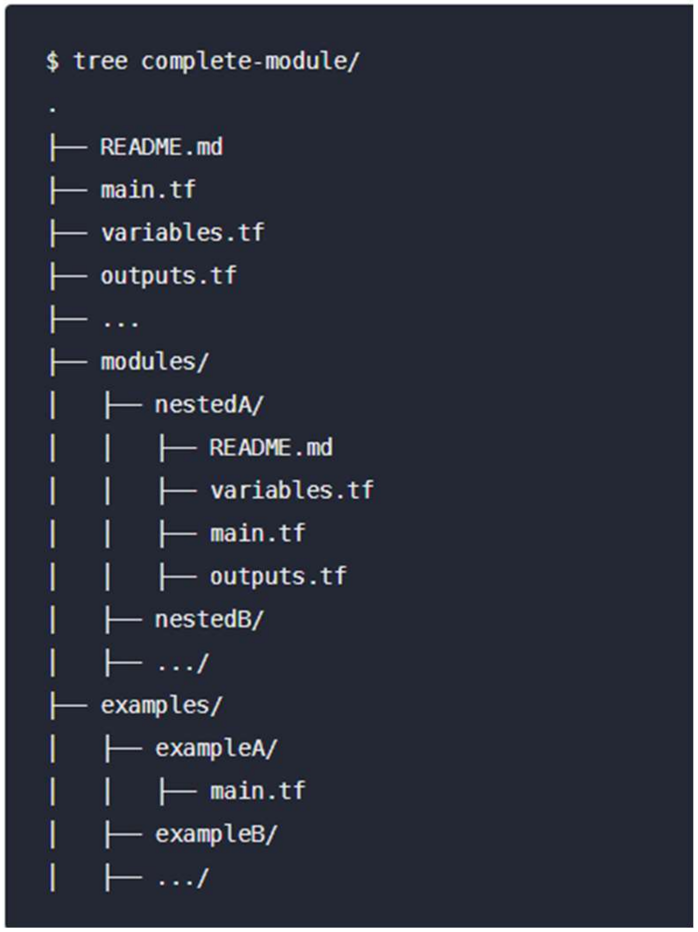
```
module "vpc" {
  source = "git::https://example.com/vpc.git"
}

module "storage" {
  source = "git::ssh://username@example.com/storage.git"
}
```

Terraform modules

- **Standard Module Structure**



```
$ tree minimal-module/
.
├── README.md
├── main.tf
├── variables.tf
├── outputs.tf
```



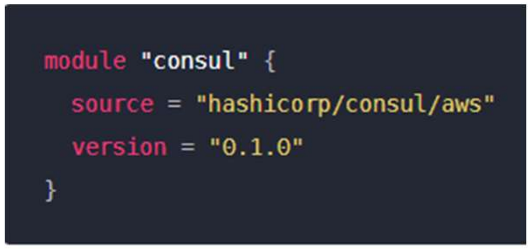
```
$ tree complete-module/
.
├── README.md
├── main.tf
├── variables.tf
├── outputs.tf
├── ...
├── modules/
│   ├── nestedA/
│   │   ├── README.md
│   │   ├── variables.tf
│   │   ├── main.tf
│   │   ├── outputs.tf
│   ├── nestedB/
│   ├── .../
├── examples/
│   ├── exampleA/
│   │   ├── main.tf
│   ├── exampleB/
│   ├── .../
```

Terraform modules

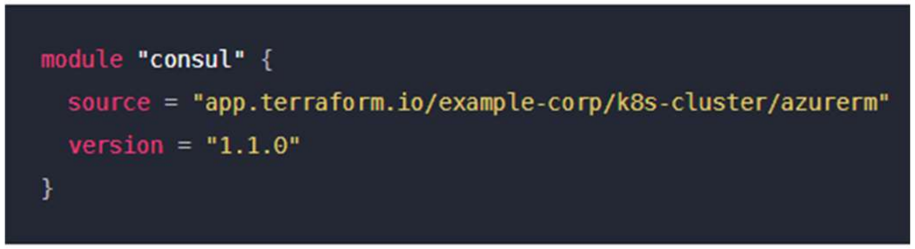
Terraform Registry

- A module registry is the native way of distributing Terraform modules for use across multiple configurations, using a Terraform-specific protocol that has full support for module versioning.

- Terraform Registry is an index of modules shared publicly using this protocol. This public registry is the easiest way to get started with Terraform and find modules created by others in the community.



```
module "consul" {
  source = "hashicorp/consul/aws"
  version = "0.1.0"
}
```



```
module "consul" {
  source = "app.terraform.io/example-corp/k8s-cluster/azurerm"
  version = "1.1.0"
}
```

Terraform State

- Terraform must store state about your managed infrastructure and configuration.
- This state is used by Terraform to map real world resources to your configuration, keep track of metadata, and to improve performance for large infrastructures.
- This 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.
- Terraform uses this local state to create plans and make changes to your infrastructure. Prior to any operation, Terraform does a refresh to update the state with the real infrastructure.
- The primary purpose of Terraform state is to store bindings between objects in a remote system and resource instances declared in your configuration

```
{
  "terraform_version": "0.12.7",
  "serial": 14,
  "lineage": "452b4191-89f6-db17-a3b1-4470dcb00607",
  "outputs": {
    "catapp_url": {
      "value": "http://go-hashicat-5c0265179ccda553.workshop.aws.hashidemos.io",
      "type": "string"
    },
  }
}
```

Terraform State

Purpose of Terraform State

- Mapping to the Real World

  Terraform requires some sort of database to map Terraform config to the real world. When you have a resource "aws_instance" "foo" in your configuration, Terraform uses this map to know that instance i-abcd1234 is represented by that resource.

- Metadata

  Terraform typically uses the configuration to determine dependency order. However, when you delete a resource from a Terraform configuration, Terraform must know how to delete that resource. Terraform can see that a mapping exists for a resource not in your configuration and plan to destroy.

- Performance

  In addition to basic mapping, Terraform stores a cache of the attribute values for all resources in the state. This is the most optional feature of Terraform state and is done only as a performance improvement.

- Syncing

  With a fully-featured state backend, Terraform can use remote locking as a measure to avoid two or more different users accidentally running Terraform at the same time, and thus ensure that each Terraform run begins with the most recent updated state.

Terraform State

Manipulating State

- The terraform state list command shows the resource addresses for every resource Terraform knows about in a configuration, optionally filtered by partial resource address.
- The terraform state show command displays detailed state data about one resource.
- The terraform refresh command updates state data to match the real-world condition of the managed resources. This is done automatically during plans and applies, but not when interacting with state directly.

```
$ terraform state list aws_instance.bar
aws_instance.bar[0]
aws_instance.bar[1]
```

```
$ terraform state show 'packet_device.worker'
# packet_device.worker:
resource "packet_device" "worker" {
    billing_cycle = "hourly"
    created       = "2015-12-17T00:06:56Z"
    facility      = "ewr1"
    hostname      = "prod-xyz01"
    id            = "6015bg2b-b8c4-4925-aad2-f0671d5d3b13"
    locked        = false
}
```

```
terraform apply -refresh-only -auto-approve
```

Terraform Settings

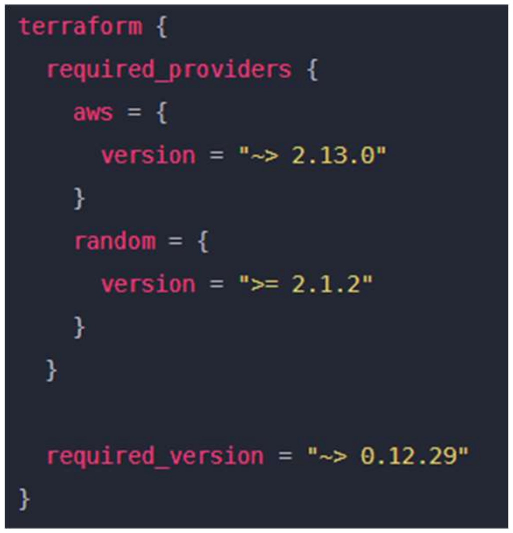
The special terraform configuration block type is used to configure some behaviors of Terraform itself, such as requiring a minimum Terraform version to apply your configuration.

Terraform Block Syntax

- Each terraform block can contain a number of settings related to Terraform's behavior.
- 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 language built-in functions.

Specifying a Required Terraform Version

Specifying Provider Requirements



```
terraform {
  required_providers {
    aws = {
      version = "~> 2.13.0"
    }
    random = {
      version = ">= 2.1.2"
    }
  }

  required_version = "~> 0.12.29"
}
```

Terraform Settings

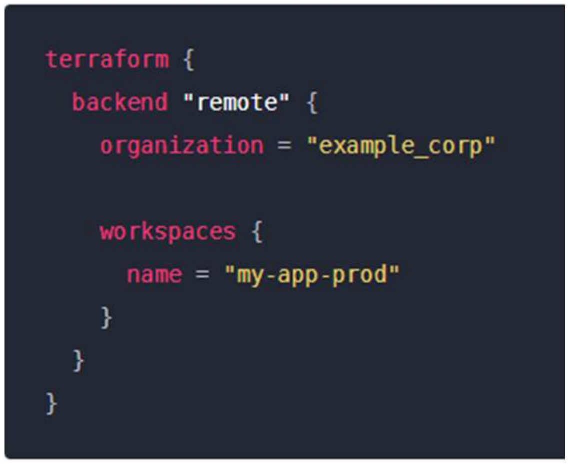
Backends

Backends define where Terraform's state snapshots are stored.

What Backends Do

- Backends primarily determine where Terraform stores its state.
- Terraform uses this persisted state data to keep track of the resources it manages.
- By default, Terraform implicitly uses a backend called local to store state as a local file on disk
- Every other backend stores state in a remote service of some kind, which allows multiple people to access it.

Using a Backend Block



```
terraform {
  backend "remote" {
    organization = "example_corp"

    workspaces {
      name = "my-app-prod"
    }
  }
}
```

Terraform Settings

Backend Types

Terraform's backends are divided into two main types, according to how they handle state and operations:

local : The local backend stores state on the local filesystem, locks that state using system APIs, and performs operations locally.

```
terraform {
  backend "local" {
    path = "relative/path/to/terraform.tfstate"
  }
}
```

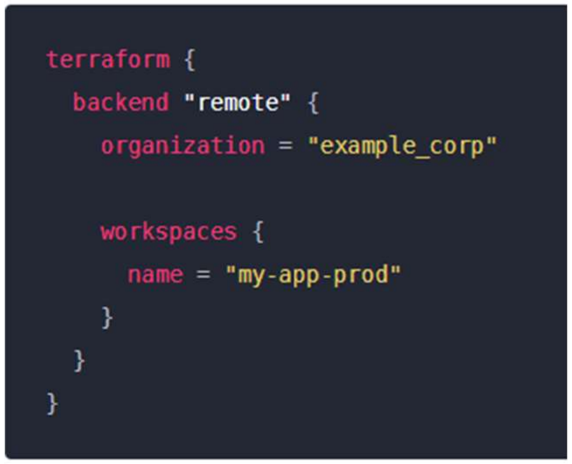
Terraform Settings

remote:

The remote backend stores Terraform state and may be used to run operations in Terraform Cloud.

Standard Backends

- artifactory
- azurerm
- consul
- kubernetes
- s3
- etcd



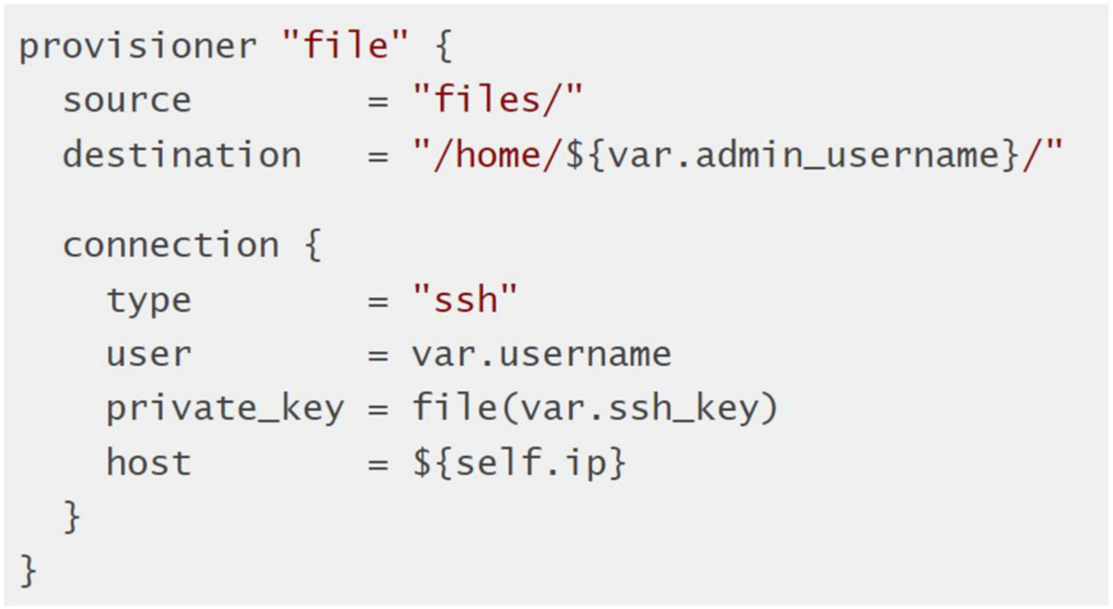
```
terraform {
  backend "remote" {
    organization = "example_corp"

    workspaces {
      name = "my-app-prod"
    }
  }
}
```

Terraform 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.

Provisioners are a Last Resort

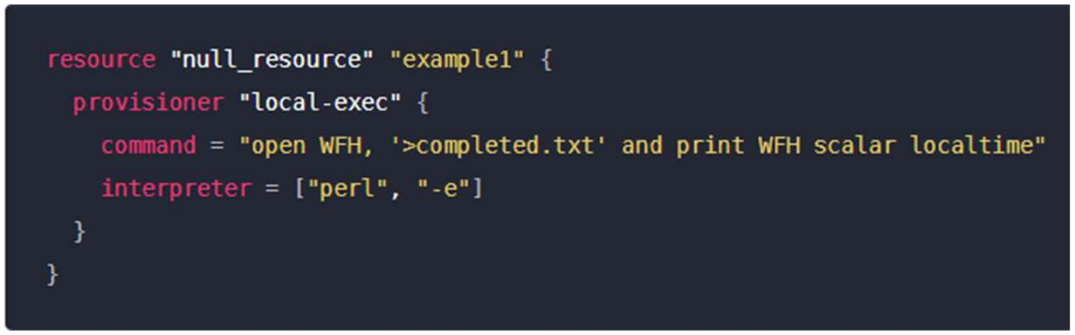
file Provisioner : used to copy files or directories from the machine executing Terraform to the newly created resource. The file provisioner supports both ssh and winrm type connections



```
provisioner "file" {
  source      = "files/"
  destination = "/home/${var.admin_username}/"

  connection {
    type        = "ssh"
    user        = var.username
    private_key = file(var.ssh_key)
    host        = ${self.ip}
  }
}
```

Terraform Provisioners

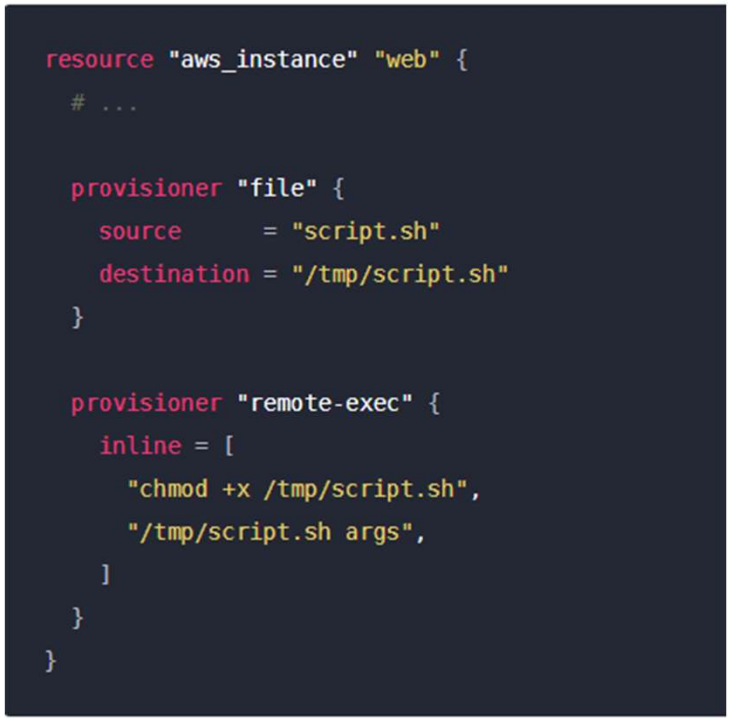
local-exec Provisioner: invokes a local executable after a resource is created. This invokes a process on the machine running Terraform, not on the resource



```
resource "null_resource" "example1" {
  provisioner "local-exec" {
    command = "open WFH, '>completed.txt' and print WFH scalar localtime"
    interpreter = ["perl", "-e"]
  }
}
```

Terraform Provisioners

remote-exec Provisioner: invokes a script on a remote resource after it is created. This can be used to run a configuration management tool, bootstrap into a cluster, etc

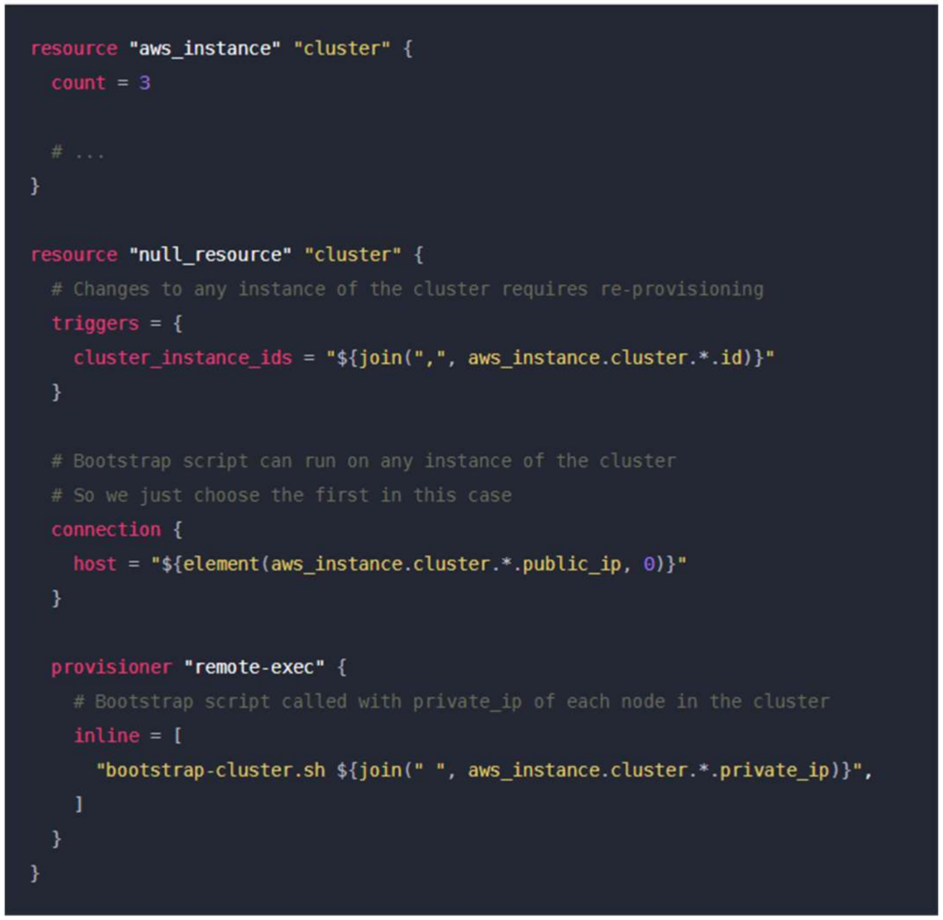


```
resource "aws_instance" "web" {
  # ...

  provisioner "file" {
    source      = "script.sh"
    destination = "/tmp/script.sh"
  }

  provisioner "remote-exec" {
    inline = [
      "chmod +x /tmp/script.sh",
      "/tmp/script.sh args",
    ]
  }
}
```

Terraform Provisioners

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.



```
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)}"
  }

  # Bootstrap script can run on any instance of the cluster
  # So we just choose the first in this case
  connection {
    host = "${element(aws_instance.cluster.*.public_ip, 0)}"
  }

  provisioner "remote-exec" {
    # Bootstrap script called with private_ip of each node in the cluster
    inline = [
      "bootstrap-cluster.sh ${join(" ", aws_instance.cluster.*.private_ip)}",
    ]
  }
}
```

Terraform Format

Terraform comes with a built in code formatter/cleaner. It can make all your margins and list indentation neat and tidy. Beauty works better.

terraform fmt

Simply run it in a directory containing *.tf files and it will tidy up your code for you.

Terraform validate

The terraform validate command validates the configuration files in a directory, referring only to the configuration and not accessing any remote services such as remote state, provider APIs, etc.

Validate runs checks that verify whether a configuration is syntactically valid and internally consistent, regardless of any provided variables or existing state

terraform validate

Terraform workspaces

workspaces are separate instances of state data that can be used from the same working directory. You can use workspaces to manage multiple non-overlapping groups of resources with the same configuration.

Every initialized working directory has at least one workspace

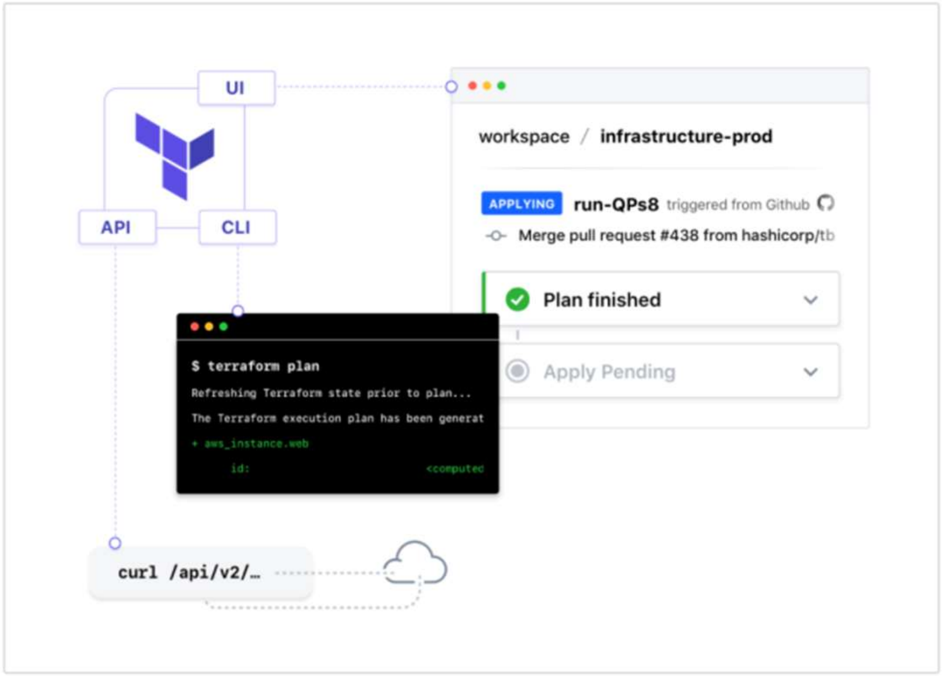
The ***terraform workspace*** command is used to manage workspaces.

- ***terraform workspace list***
- ***terraform workspace select***
- ***terraform workspace new***
- ***terraform workspace delete***
- ***terraform workspace show***

Terraform Cloud

Terraform Cloud is a free to use SaaS application that provides the best workflow for writing and building infrastructure as code with Terraform.

- State storage and management
- Web UI for viewing and approving Terraform
- Private module registry
- Version Control System (VCS) integration
- CLI, API or GUI driven actions
- Notifications for run events
- Full HTTP API for automation


UI
API
CLI
workspace / infrastructure-prod
APPLYING
run-QPs8 triggered from Github
Merge pull request #438 from hashicorp/tb
Plan finished
Apply Pending

```
$ terraform plan
Refreshing Terraform state prior to plan...
The Terraform execution plan has been generat
+ aws_instance.web
      id:                              <computed
```

```
curl /api/v2/…
```

Terraform Cloud or Terraform Enterprise?

Terraform Cloud is a hosted application that provides features like remote state management, API driven runs, policy management and more. Many users prefer a cloud-based SaaS solution because they don't want to maintain the infrastructure to run it.

Terraform Cloud for Business utilizes the same hosted environment as Terraform Cloud, but you get the features more applicable to larger teams. Single Sign-on, Audit Logging, and the ability to Terraform on-prem resources from the cloud.

Terraform Enterprise is the same application, but it runs in your own cloud environment or data center. Some users require more control over the Terraform Cloud application, or wish to run it in restricted networks behind corporate firewalls.

The feature list for these offerings is nearly identical.

Integrate Jenkins & Terraform