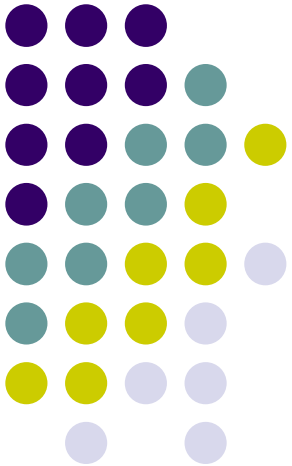
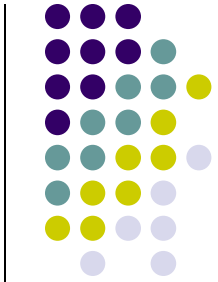
HashiCorp Certified: Terraform Associate

Raj



Configuration Management Vs Infrastructure Orchestration

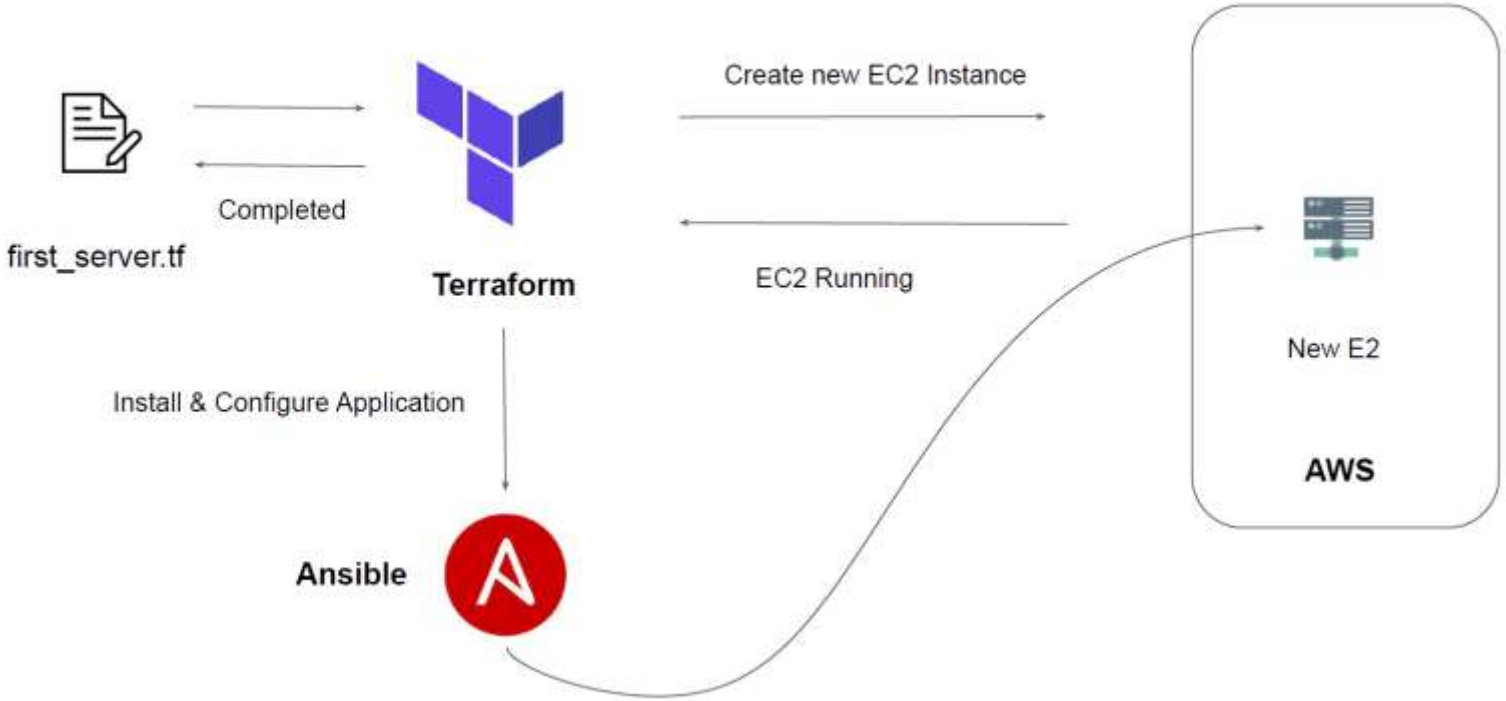


Ansible, Chef, Puppet are configuration management tools which means that they are primarily designed to install and manage software on existing servers.

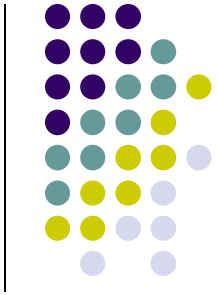
Terraform, CloudFormation are the infrastructure orchestration tools which basically means they can provision the servers and infrastructure by themselves.

Configuration Management tools can do some degree of infrastructure provisioning, but the focus here is that some tools are going to be better fit for certain type of tasks.

IAC & Configuration Management = Friends



Choice of tools



Question remains on how to choose right IAC tool for the organization

i) Is your infrastructure going to be vendor specific in longer term ? Example AWS.

ii) Are you planning to have multi-cloud / hybrid cloud based infrastructure ?

iii) How well does it integrate with configuration management tools ?

iv) Price and Support

Terraform

i) Supports multiple platforms, has hundreds of providers.

ii) Simple configuration language and faster learning curve.

iii) Easy integration with configuration management tools like Ansible.

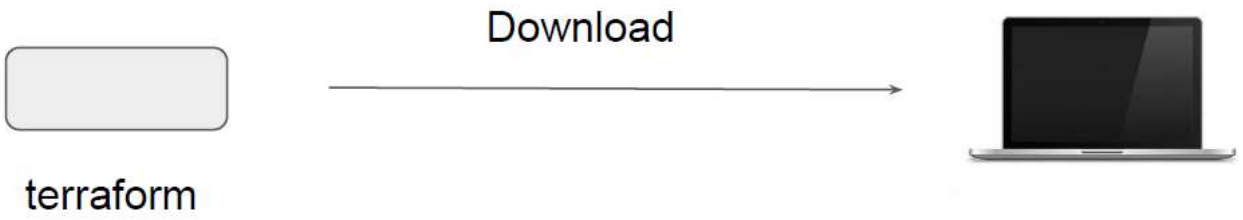
iv) Easily extensible with the help of plugins.

v) Free !!!

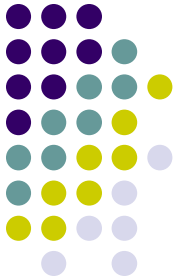
Terraform Installation

Terraform installation is very simple.

You have a single binary file, download and use it.



Terraform support



Terraform works on multiple platforms, these includes:

- Windows
- macOS
- Linux
- FreeBSD
- OpenBSD
- Solaris

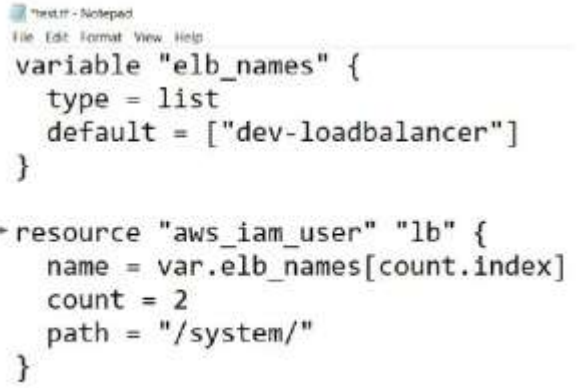
Terraform editor

You can write Terraform code in Notepad and it will not have any impact.

Downsides:

- Slower Development
- Limited Features

HashiCorp
Terraform

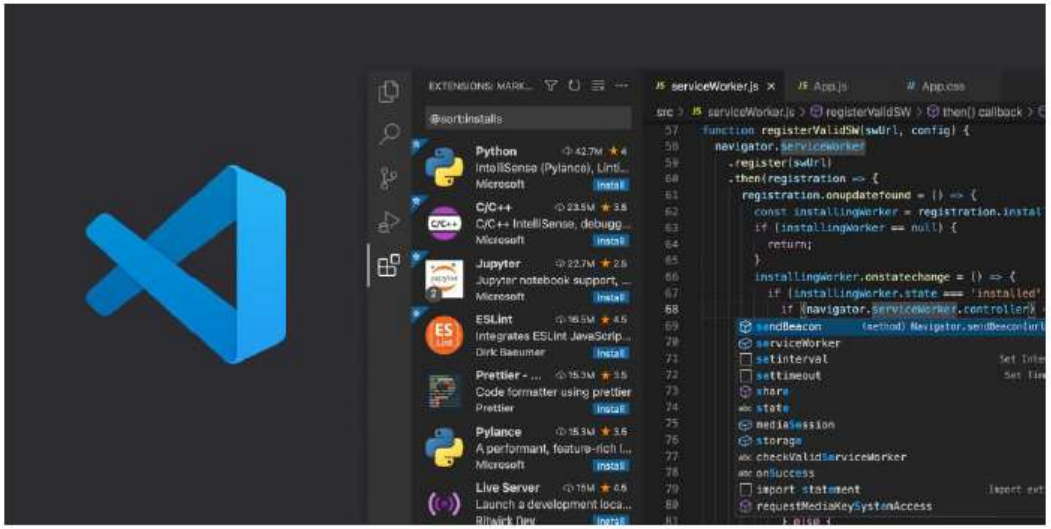


Terraform Editor

We are going to make use of Visual Studio Code as primary editor in this course.

Advantages:

1. Supports Windows, Mac, Linux
2. Supports Wide variety of programming languages.
3. Many Extensions.



Visual studio extensions

Extensions are add-ons that allow you to customize and enhance your experience in Visual Studio by adding new features or integrating existing tools

They offer wide range of functionality related to colors, auto-complete, report spelling errors etc.

Terraform extension

HashiCorp also provides extension for Terraform for Visual Studio Code.

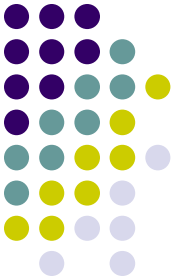
demo.tf

```
resource "aws_instance" "myec2" {
    ami = "ami-00c39f71452c08778"
    instance_type = "t2.micro"
}
```

demo.tf > ...

```
resource "aws_instance" "myec2" {
    ami = "ami-00c39f71452c08778"
    instance_type = "t2.micro"
}
```

Setting up the Lab

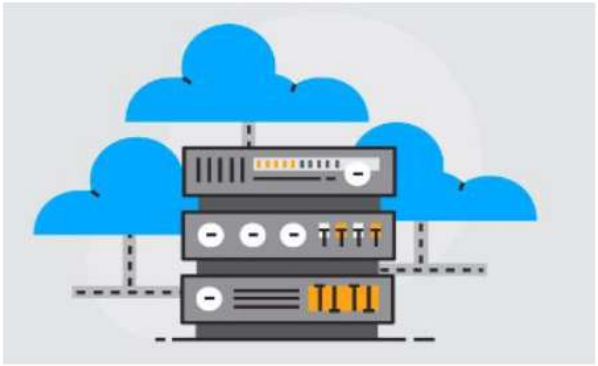


Let’s start Rolling !

Let’s start

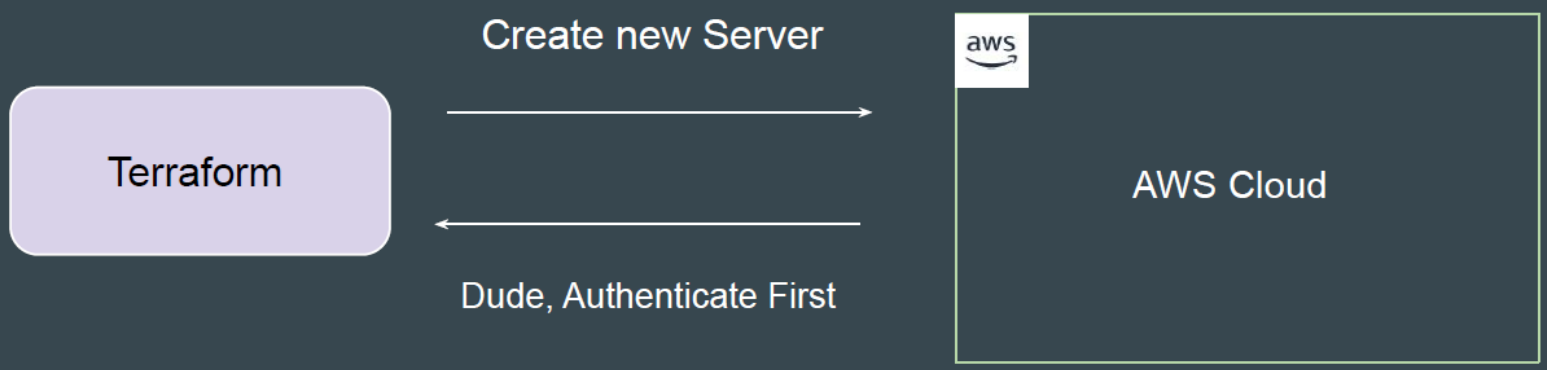
i) Create a new AWS Account.

ii) Begin the course

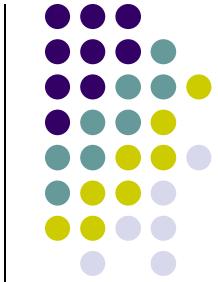


Authentication & Authorization

Before we start working on managing environments through Terraform, the first important step is related to Authentication and Authorization.



Basics of Authentication and Authorization



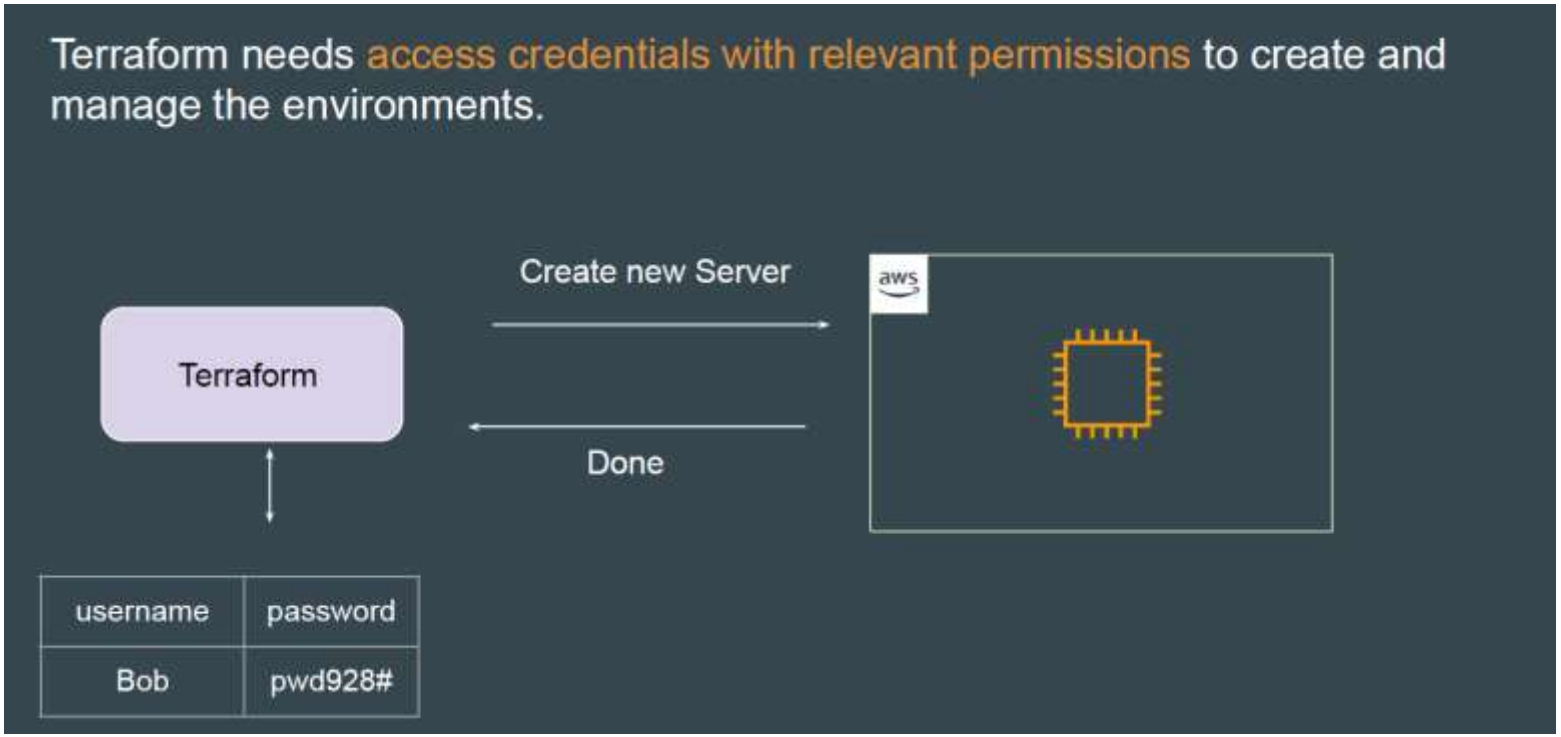
Authentication is the process of verifying who a user is.

Authorization is the process of verifying what they have access to

Example:

Alice is a user in AWS with no access to any service.

Terraform use case



Access credentials

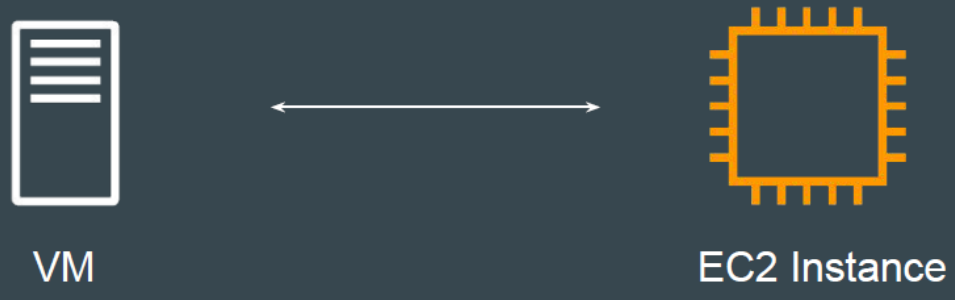
Depending on the provider, the type of access credentials would change.

Provider	Access Credentials
AWS	Access Keys and Secret Keys
GitHub	Tokens
Kubernetes	Kubeconfig file, Credentials Config
Digital Ocean	Tokens

Basics of EC2

EC2 stands for Elastic Compute Cloud.

In-short, it's a name for a virtual server that you launch in AWS.

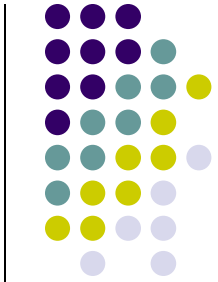


Available Regions

Cloud providers offers multiple regions in which we can create our resource.

You need to decide the region in which Terraform would create the resource.

Virtual machine configuration



A Virtual Machine would have it's own set of configurations.

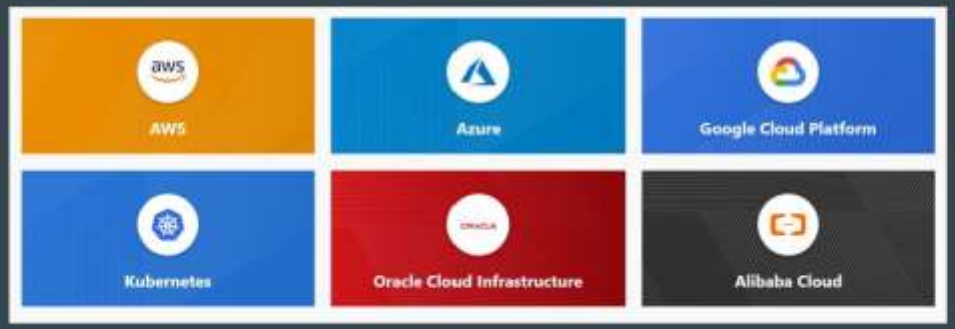
- CPU
- Memory
- Storage
- Operating System

While creating VM through Terraform, you will need to define these.

Providers

Terraform supports multiple providers.

Depending on what type of infrastructure we want to launch, we have to use appropriate providers accordingly.



Provider Plugin

A provider is a plugin that lets Terraform manage an external API.

When we run terraform init, plugins required for the provider are automatically downloaded and saved locally to a .terraform directory.

```
C:\Users\zealv\Desktop\kplabs-terraform>terraform init

Initializing the backend...

Initializing provider plugins...
- Finding latest version of hashicorp/aws...
- Installing hashicorp/aws v4.60.0...
- Installed hashicorp/aws v4.60.0 (signed by HashiCorp)

Terraform has created a lock file .terraform.lock.hcl to record the provider
selections it made above. Include this file in your version control repository
so that Terraform can guarantee to make the same selections by default when
you run "terraform init" in the future.

Terraform has been successfully initialized!

You may now begin working with Terraform. Try running "terraform plan" to see
any changes that are required for your infrastructure. All Terraform commands
should now work.

If you ever set or change modules or backend configuration for Terraform,
rerun this command to reinitialize your working directory. If you forget, other
commands will detect it and remind you to do so if necessary.
```

Resource

Resource block describes one or more infrastructure objects

Example:

- resource aws_instance
- resource aws_alb
- resource iam_user
- resource digitalocean_droplet

```
resource "aws_instance" "myec2" {
   ami = "ami-082b5a644766e0e6f"
   instance_type = "t2.micro"
}
```

Resource Block

A resource block declares a resource of a given type ("aws_instance") with a given local name ("myec2").

Resource type and Name together serve as an identifier for a given resource and so must be unique.

```
resource "aws_instance" "myec2" {
   ami = "ami-082b5a644766e0e6f"
   instance_type = "t2.micro"
}
```

EC2 Instance Number 1

```
resource "aws_instance" "web" {
  ami           = ami-123
  instance_type = "t2.micro"
}
```

EC2 Instance Number 2

Provider and Resource

You can only use the resource that are supported by a specific provider.

In the below example, provider of Azure is used with resource of aws_instance

```
provider "azurerm" {}

resource "aws_instance" "web" {
  ami           = ami-123
  instance_type = "t2.micro"

}
```

Across provider syntax - cool

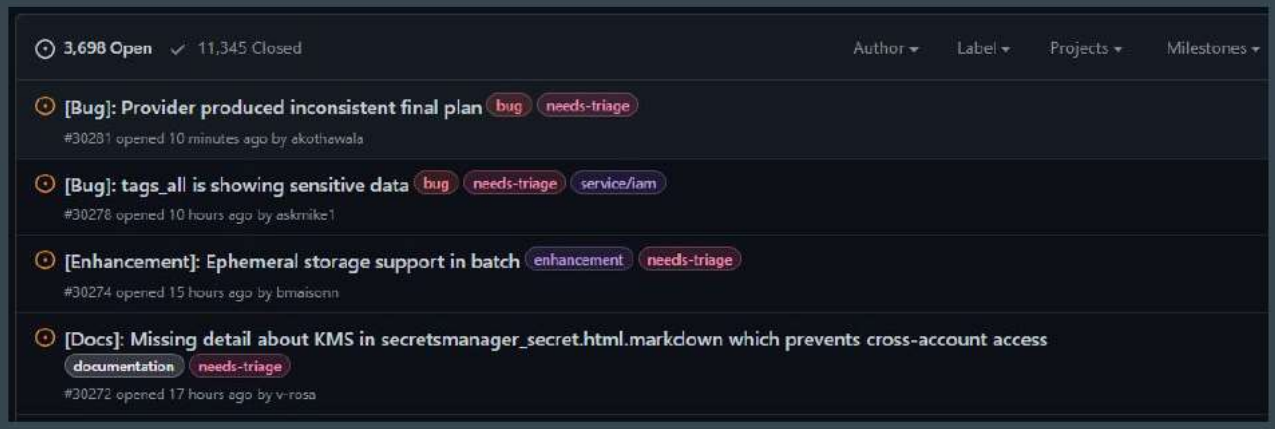
The core concepts, standard syntax remains similar across all providers.

If you learn the basics, you should be able to work with all providers easily.

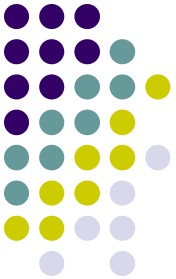
Reporting bug with Provider

A provider that is maintained by HashiCorp does not mean it has no bugs.

It can happen that there are inconsistencies from your output and things mentioned in documentation. You can raise issue at Provider page.



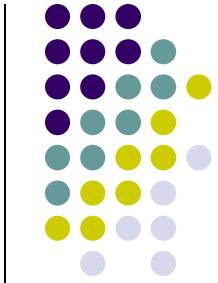
Provider Maintainers



There are 3 primary type of provider tiers in Terraform.

Provider Tiers	Description
Official	Owned and Maintained by HashiCorp.
Partner	Owned and Maintained by Technology Company that maintains direct partnership with HashiCorp.
Community	Owned and Maintained by Individual Contributors.

Provider Namespace



Namespaces are used to help users identify the organization or publisher responsible for the integration

Tier	Description
Official	hashicorp
Partner	Third-party organization e.g. mongodb/mongodbatlas
Community	Maintainer's individual or organization account, e.g. DeviaVir/gsuite

Explicit – outside HCL

Terraform requires explicit source information for any providers that are not HashiCorp-maintained, using a new syntax in the required_providers nested block inside the terraform configuration block

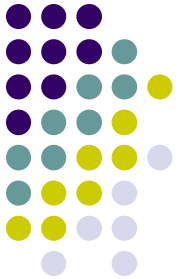
```
provider "aws" {
  region     = "us-west-2"
  access_key = "PUT-YOUR-ACCESS-KEY-HERE"
  secret_key = "PUT-YOUR-SECRET-KEY-HERE"
}
```

HashiCorp Maintained

```
terraform {
  required_providers {
    digitalocean = {
      source = "digitalocean/digitalocean"
    }
  }
}

provider "digitalocean" {
  token = "PUT-YOUR-TOKEN-HERE"
}
```

Non-HashiCorp Maintained

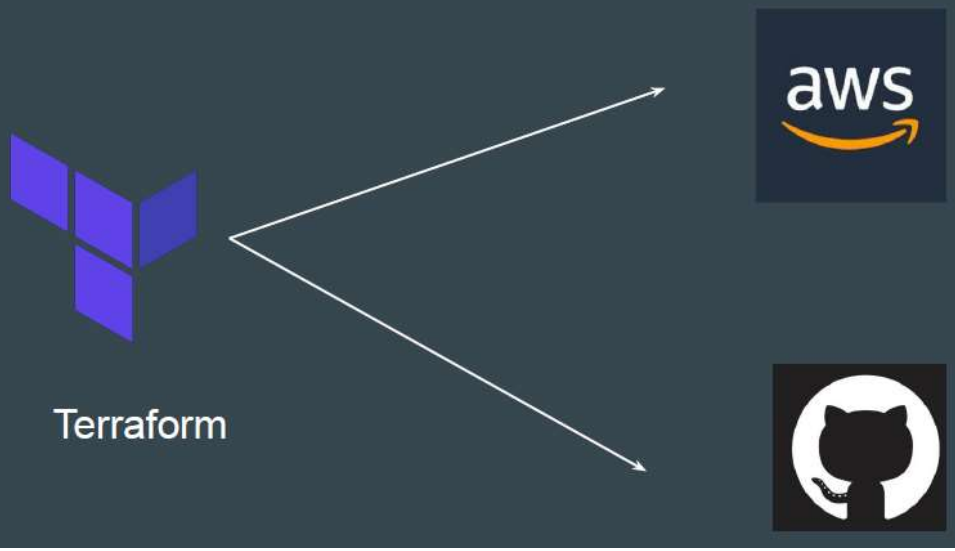
Day 2



Destroy

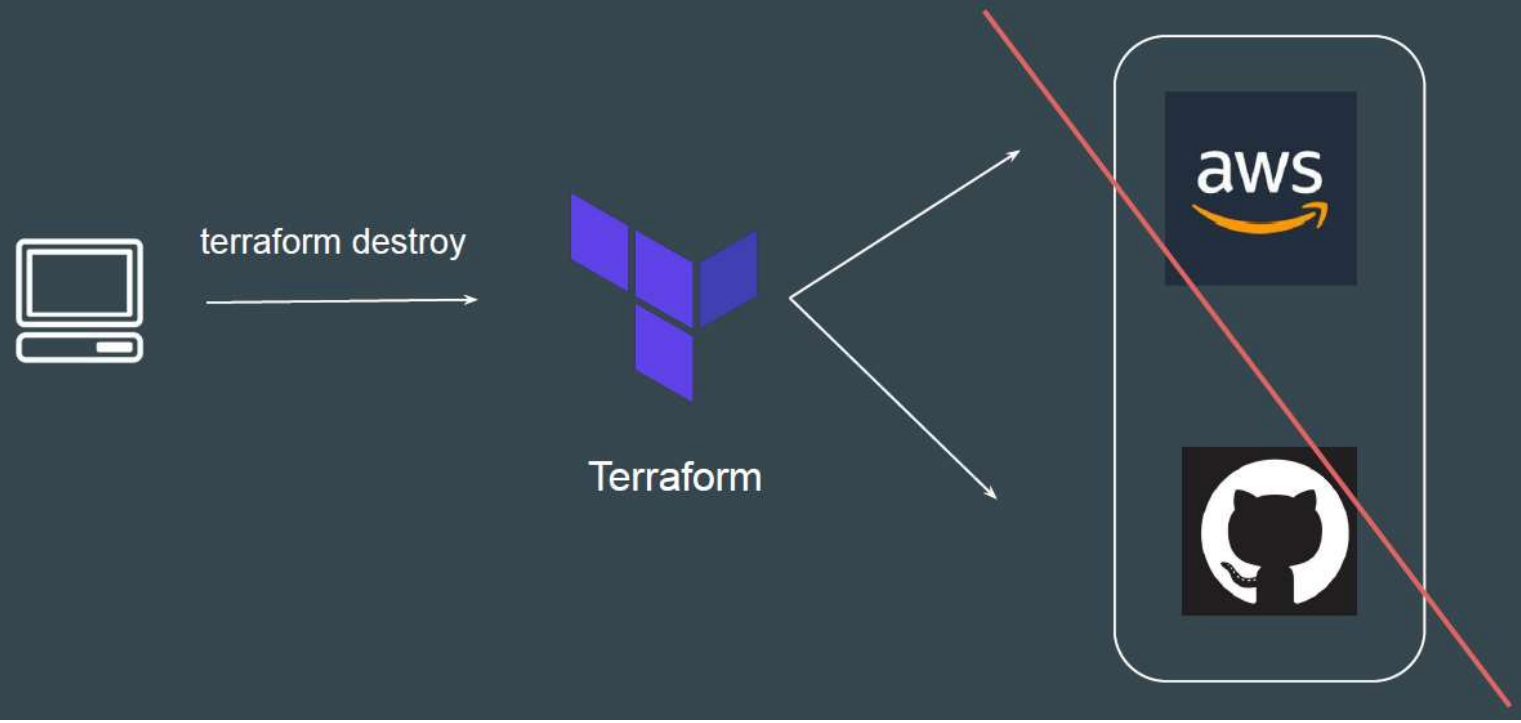
If you keep the infrastructure running, you will get charged for it.

Hence it is important for us to also know on how we can delete the infrastructure resources created via terraform.



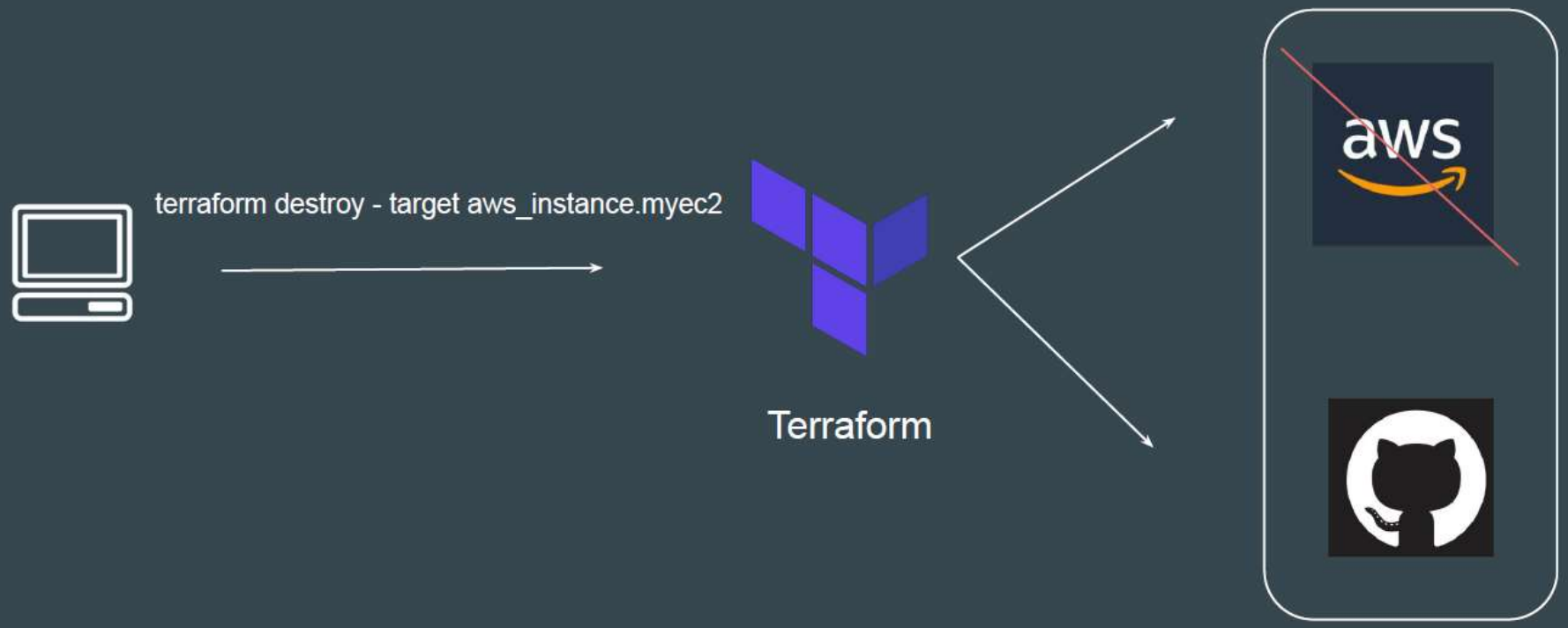
Destroy all

terraform destroy allows us to destroy all the resource that are created within the folder.

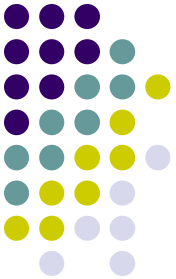


Destroy some

terraform destroy with -target flag allows us to destroy specific resource.



Destroy with Target



The -target option can be used to focus Terraform's attention on only a subset of resources.

Combination of : Resource Type + Local Resource Name

Resource Type	Local Resource Name
aws_instance	myec2
github_repository	example

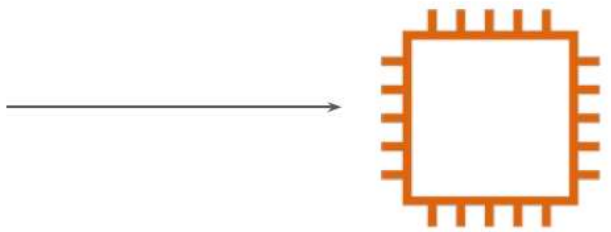
```
resource "aws_instance" "myec2" {
    ami = "ami-00c39f71452c08778"
    instance_type = "t2.micro"
}
```

```
resource "github_repository" "example" {
  name        = "example"
  description = "My awesome codebase"

  visibility = "public"

}
```

Desired state

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

```
resource "aws_instance" "myec2" {
   ami = "ami-082b5a644766e0e6f"
   instance_type = "t2.micro"
}
```



EC2 - t2.micro

Current state

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

```
resource "aws_instance" "myec2" {
   ami = "ami-082b5a644766e0e6f"
   instance_type = "t2.micro"
}
```

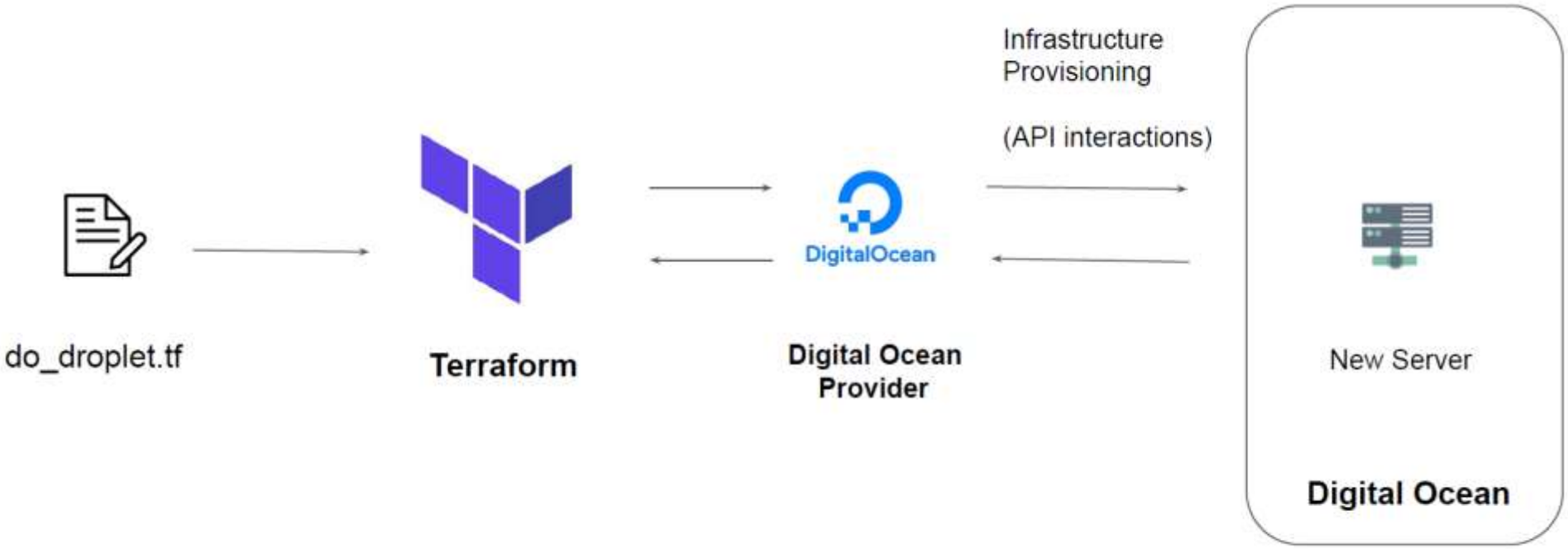
t2.medium

Roadmap towards Desired state

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

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

Provider versioning



Overview of provider versioning

Provider plugins are released separately from Terraform itself.

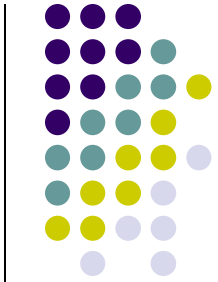
They have different set of version numbers.

.

Version 1

Version 2

Explicit provider versioning



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

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

```
terraform {
  required_providers {
    aws = {
      source  = "hashicorp/aws"
      version = "~> 3.0"
    }
  }
}

provider "aws" {
  region = "us-east-1"
}
```

Arguments for specific provider

There are multiple ways for specifying the version of a provider.

Version Number Arguments	Description
>=1.0	Greater than equal to the version
<=1.0	Less than equal to the version
~>2.0	Any version in the 2.X range.
>=2.10,<=2.30	Any version between 2.10 and 2.30

Dependency lock file

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

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

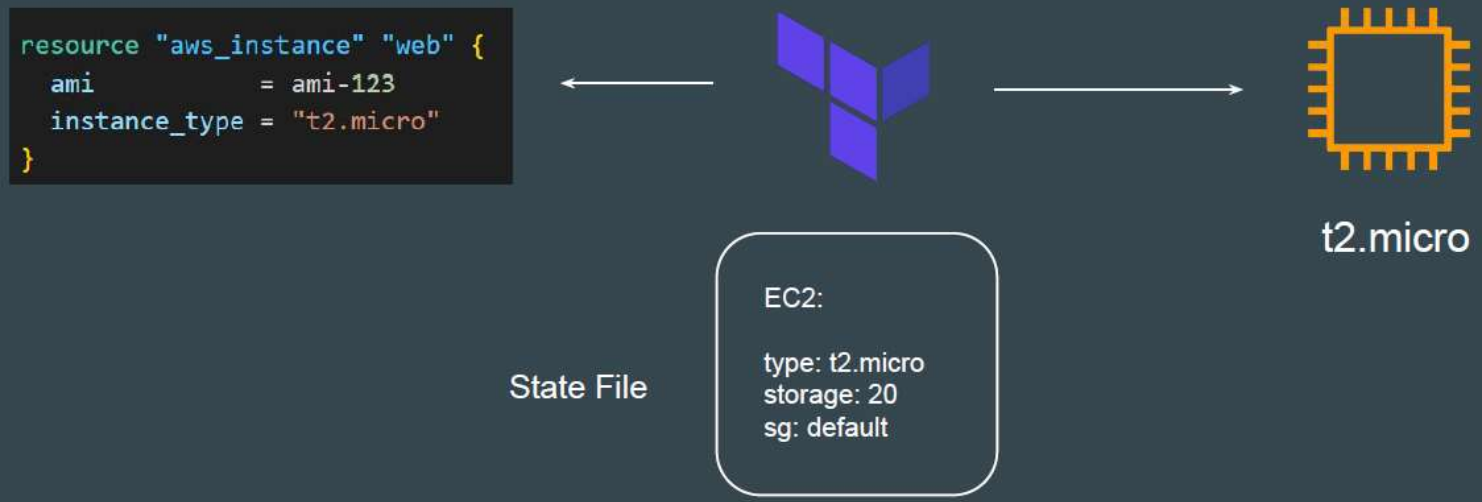
You can override that behavior by adding the -upgrade option when you run terraform init,

```
provider "registry.terraform.io/hashicorp/aws" {
  version     = "2.70.0"
  constraints = ">= 2.31.0, <= 2.70.0"
  hashes = [
    "h1:fx8tbGVwK1YIDI6UdHLnorC9PA1ZPSWEeW3V3aDCdWY=",
    "zh:01a5f351146434b418f9ff8d8cc956ddc801110f1cc8b139e01be2ff8c544605",
    "zh:1ec08abbaf09e3e0547511d48f77a1e2c89face2d55886b23f643011c76cb247",
    "zh:606d134fef7c1357c9d155aadbee6826bc22bc0115b6291d483bc1444291c3e1",
    "zh:67e31a71a5ecbbc96a1a6708c9cc300bbfe921c322320cdbb95b9002026387e1",
```

Terraform Refresh Challenge

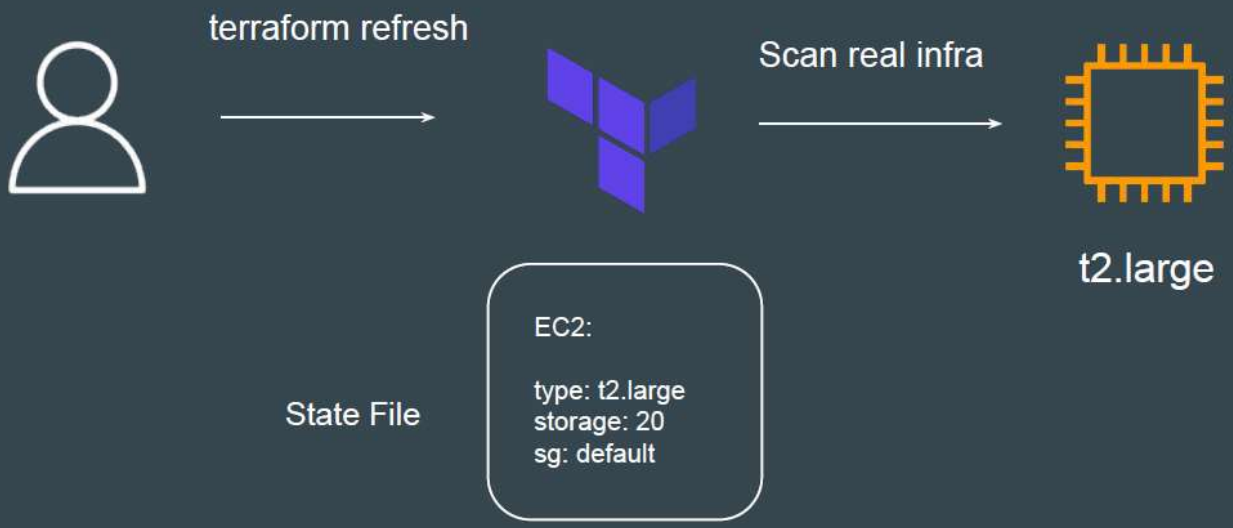
Terraform can create an infrastructure based on configuration you specified.

It can happen that the infrastructure gets modified manually.

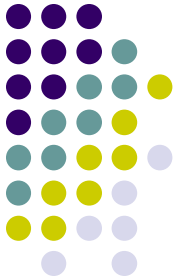


Challenge

The terraform refresh command will check the latest state of your infrastructure and update the state file accordingly.

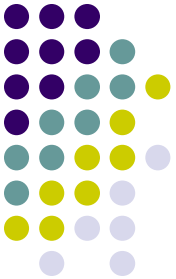


Challenge



You shouldn't typically need to use this command, because Terraform automatically performs the same refreshing actions as a part of creating a plan in both the terraform plan and terraform apply commands.

Refresh-only



The terraform refresh command is deprecated in newer version of terraform.

The -refresh-only option for terraform plan and terraform apply was introduced in Terraform v0.15.4.

Provider basics

At this stage, we have been manually hardcoding the access / secret keys within the provider block.

Although a working solution, but it is not optimal from security point of view.

aws-provider-config.tf > ...

```
provider "aws" {
  region     = "us-east-1"
  access_key = "AKIAQTSHLD4OASDZ5QJI"
  secret_key = "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"
}

resource "aws_eip" "lb" {
  domain   = "vpc"
}
```

Better way

We want our code to run successfully without hardcoding the secrets in the provider block.

aws-provider-config.tf > ...

```
provider "aws" {
  region     = "us-east-1"
}

resource "aws_eip" "lb" {
  domain   = "vpc"
}
```

Better Approach

The AWS Provider can source credentials and other settings from the shared configuration and credentials files.

aws-provider-config.tf > ...

```
provider "aws" {
  shared_config_files      = ["/Users/tf_user/.aws/conf"]
  shared_credentials_files = ["/Users/tf_user/.aws/creds"]
  profile                  = "customprofile"
}

resource "aws_eip" "lb" {
  domain   = "vpc"
}
```

Default configuration

If shared files lines are not added to provider block, by default, Terraform will locate these files at $HOME/.aws/config and $HOME/.aws/credentials on Linux and macOS.

"%USERPROFILE%\.aws\config" and "%USERPROFILE%\.aws\credentials" on Windows.

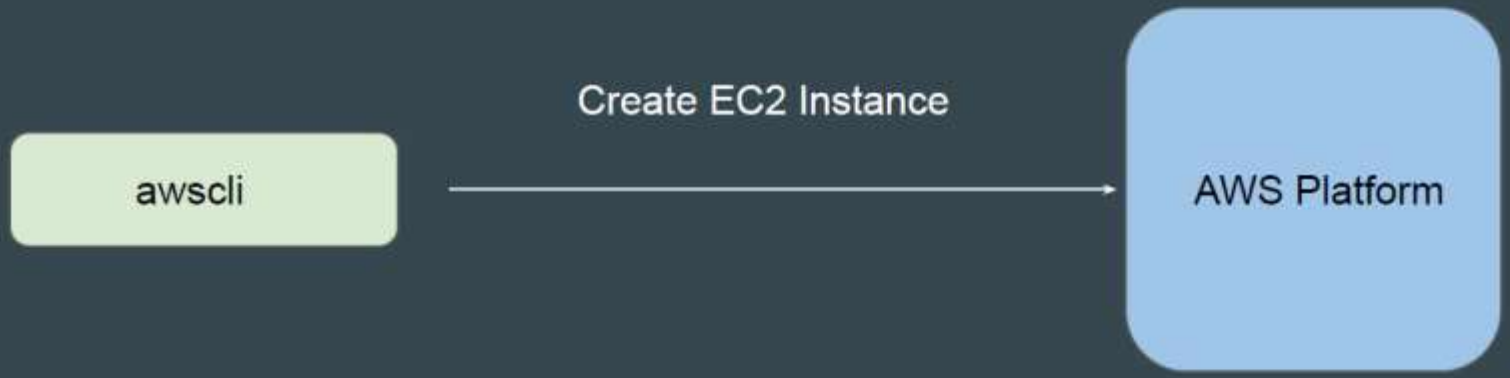
aws-provider-config.tf > ...

```
provider "aws" {
  region     = "us-east-1"
}

resource "aws_eip" "lb" {
  domain   = "vpc"
}
```

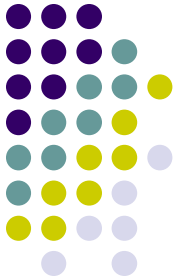
AWS CLI

AWS CLI allows customers to manage AWS resources directly from CLI.

When you configure Access/Secret keys in AWS CLI, the location in which these credentials are stored is the same default location that Terraform searches the credentials from.



Cross-Resource Attribute Reference



It can happen that in a single terraform file, you are defining two different resources.

However Resource 2 might be dependent on some value of Resource 1.

Elastic IP Address

Security group

Allow 443 from Elastic IP

Understanding workflow

eip.tf > ...

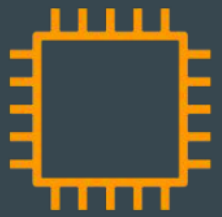
```
resource "aws_eip" "lb" {
  domain   = "vpc"
}
```

```
resource "aws_security_group" "allow_tls" {
  name        = "allow_tls"
  description = "Allow TLS inbound traffic"

  ingress {
    description      = "TLS from VPC"
    from_port        = 443
    to_port          = 443
    protocol         = "tcp"
    cidr_blocks      = [HOW-TO-ADD-ELASTIC-IP-ADDRESS-HERE?]
  }
```

Attributes

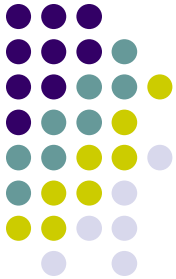
Each resource has its associated set of attributes.

Attributes are the fields in a resource that hold the values that end up in state.



Attributes	Values
ID	i-abcd
public_ip	52.74.32.50
private_ip	172.31.10.50
private_dns	ip-172-31-10-50-.ec2.internal

Cross referencing resource attribute



Terraform allows us to reference the attribute of one resource to be used in a different resource.

Elastic IP

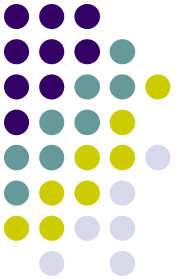
Attribute	Value
public_ip	52.72.52.72

```
resource "aws_security_group" "allow_tls" {
  name        = "allow_tls"
  description = "Allow TLS inbound traffic"
  vpc_id      = aws_vpc.main.id

  ingress {
    description      = "TLS from VPC"
    from_port        = 443
    to_port          = 443
    protocol         = "tcp"
    cidr_blocks      = [aws_eip.myeip.public_ip]

  }
```

Output values

Output values make information about your infrastructure available on the command line, and can expose information for other Terraform configurations to use.

Output value use case

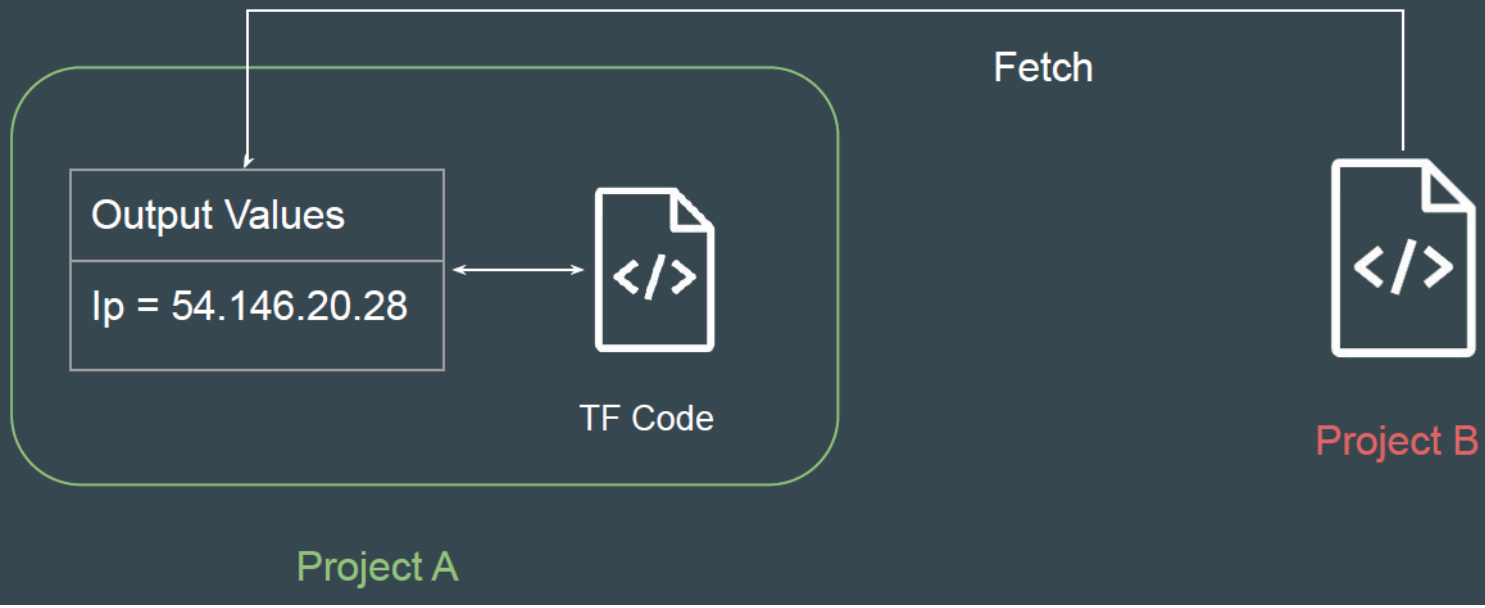


Create a Elastic IP (Public IP) resource in AWS and output the value of the EIP.

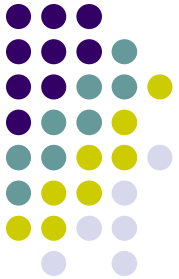
```
Plan: 1 to add, 0 to change, 0 to destroy.

Changes to Outputs:
  + demo = (known after apply)
aws_eip.lb: Creating...
aws_eip.lb: Creation complete after 3s [id=eipalloc-0680508decfe8c252]

Apply complete! Resources: 1 added, 0 changed, 0 destroyed.

Outputs:

demo = "54.146.20.18"
```

Output Project A to B

Output values defined in Project A can be referenced from code in Project B as well.

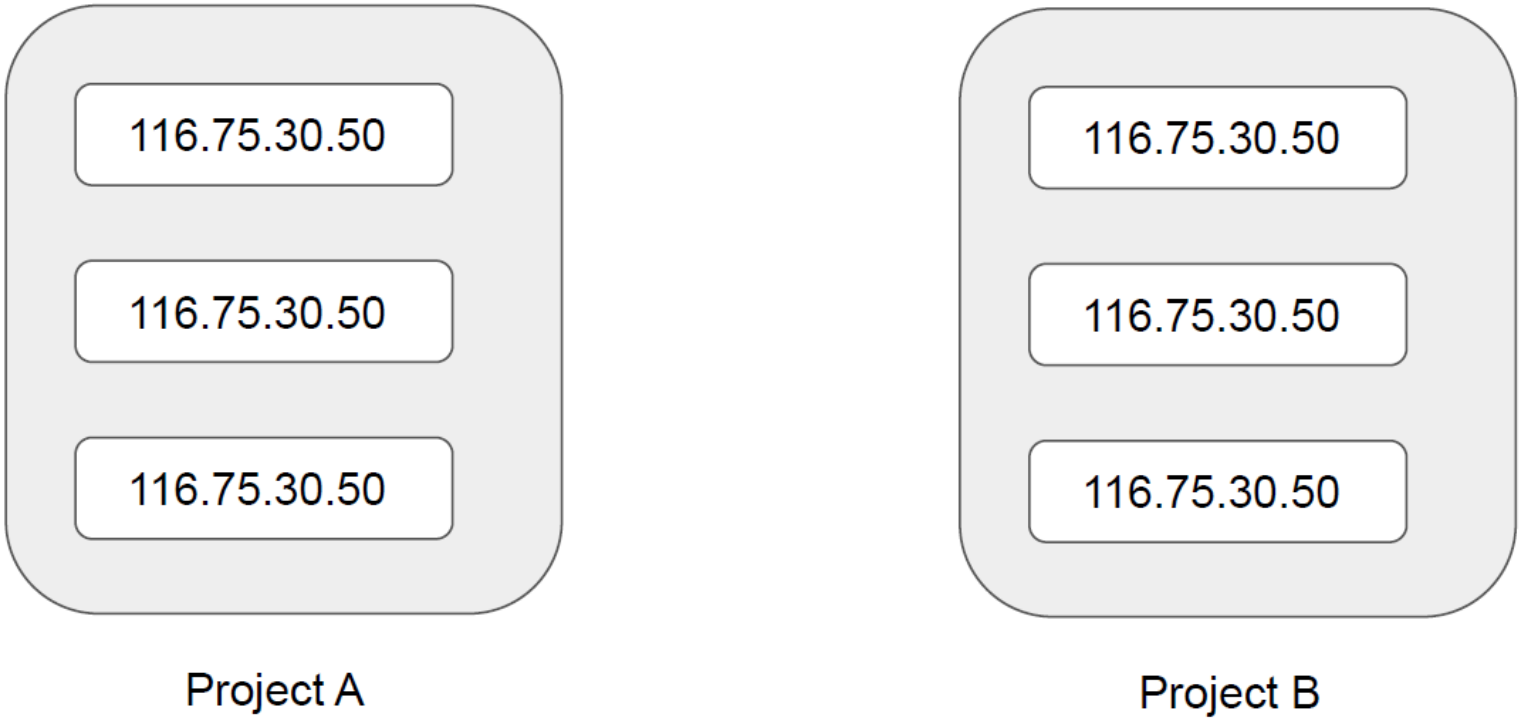


Day3



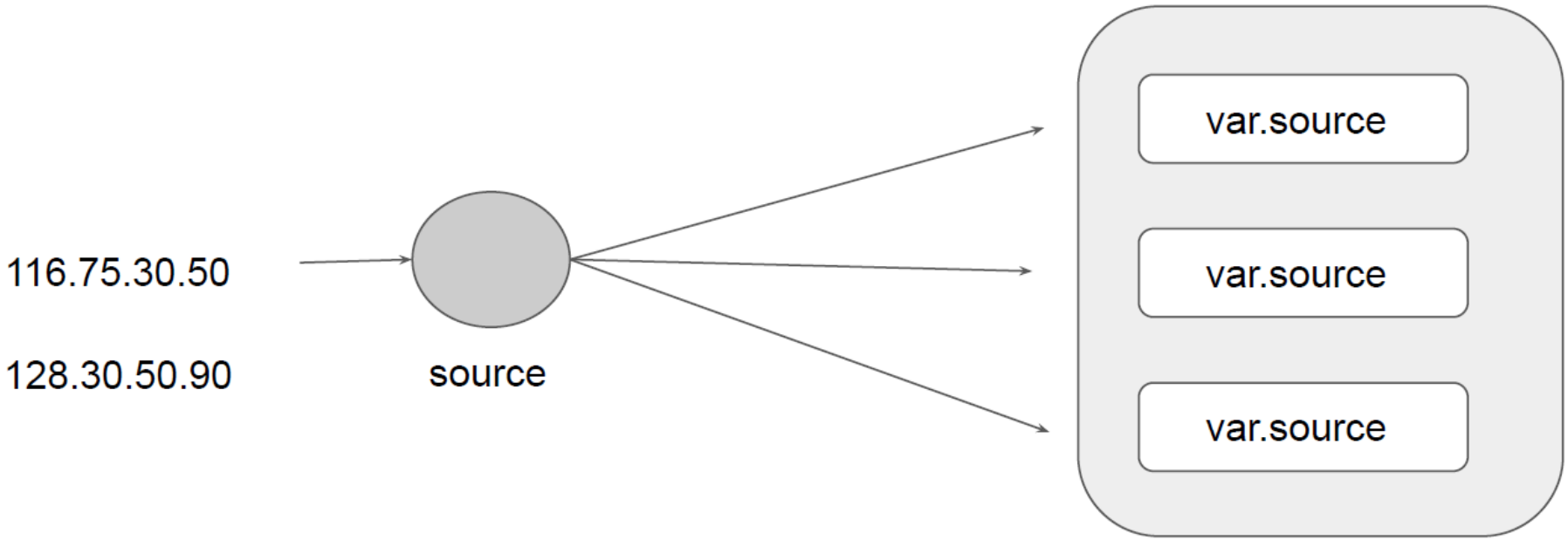
Static=work

Repeated static values can create more work in the future.

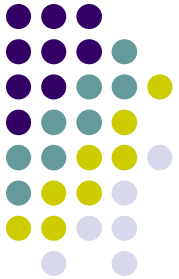


Value – Central location

We can have a central source from which we can import the values from.



Variables assignment

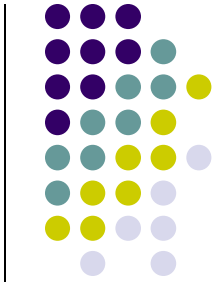


Variables in Terraform can be assigned values in multiple ways.

Some of these include:

- Environment variables
- Command Line Flags
- From a File
- Variable Defaults

Variable Type constraint

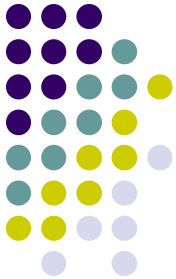


The type argument in a variable block allows you to restrict the type of value that will be accepted as the value for a variable

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

If no type constraint is set then a value of any type is accepted.

Variable

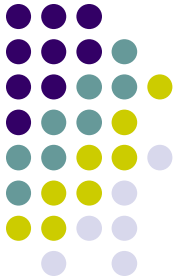


Every employee in Medium Corp is assigned a Identification Number.

Any resource that employee creates should be created with the name of the identification number only.

variables.tf	terraform.tfvars
variable “instance_name” {}	instance_name=”john-123”

Data types



Type Keywords	Description
string	Sequence of Unicode characters representing some text, like "hello".
list	Sequential list of values identified by their position. Starts with 0 [“mumbai” ,”singapore”, ”usa”]
map	a group of values identified by named labels, like {name = "Mabel", age = 52}.
number	Example: 200

Count parameter

The count parameter on resources can simplify configurations and let you scale resources by simply incrementing a number.

Let's assume, you need to create two EC2 instances. One of the common approach is to define two separate resource blocks for aws_instance.

```
resource "aws_instance" "instance-1" {
   ami = "ami-082b5a644766e0e6f"
   instance_type = "t2.micro"
}
```

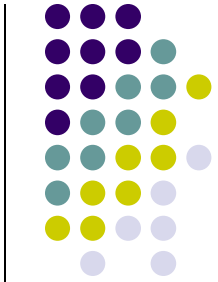
```
resource "aws_instance" "instance-2" {
   ami = "ami-082b5a644766e0e6f"
   instance_type = "t2.micro"
}
```

Count parameter

With count parameter, we can simply specify the count value and the resource can be scaled accordingly.

```
resource "aws_instance" "instance-1" {
   ami = "ami-082b5a644766e0e6f"
   instance_type = "t2.micro"
   count = 5
}
```

Count Index

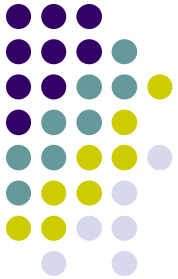


In resource blocks where count is set, an additional count object is available in expressions, so you can modify the configuration of each instance.

This object has one attribute:

count.index — The distinct index number (starting with 0) corresponding to this instance.

Challenge with Count



With the below code, terraform will create 5 IAM users. But the problem is that all will have the same name.

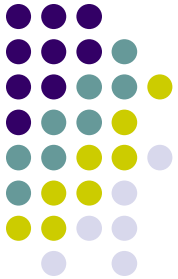
```
resource "aws_iam_user" "lb" {
  name = "loadbalancer"
  count = 5
  path = "/system/"
}
```

Count work around

count.index allows us to fetch the index of each iteration in the loop.

```
resource "aws_iam_user" "lb" {
  name = "loadbalancer.${count.index}"
  count = 5
  path = "/system/"
}
```

Default count index - solution



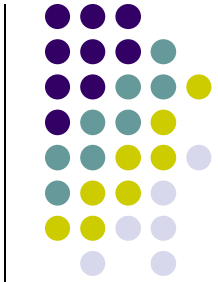
Having a username like loadbalancer0, loadbalancer1 might not always be suitable.

Better names like dev-loadbalancer, stage-loadbalancer, prod-loadbalancer is better.

count.index can help in such scenario as well.

```
variable "elb_names" {
  type        = list
  default     = ["dev-loadbalancer", "stage-loadbalancer", "prod-loadbalancer"]
}
```

Conditional expression



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

Syntax of Conditional expression:

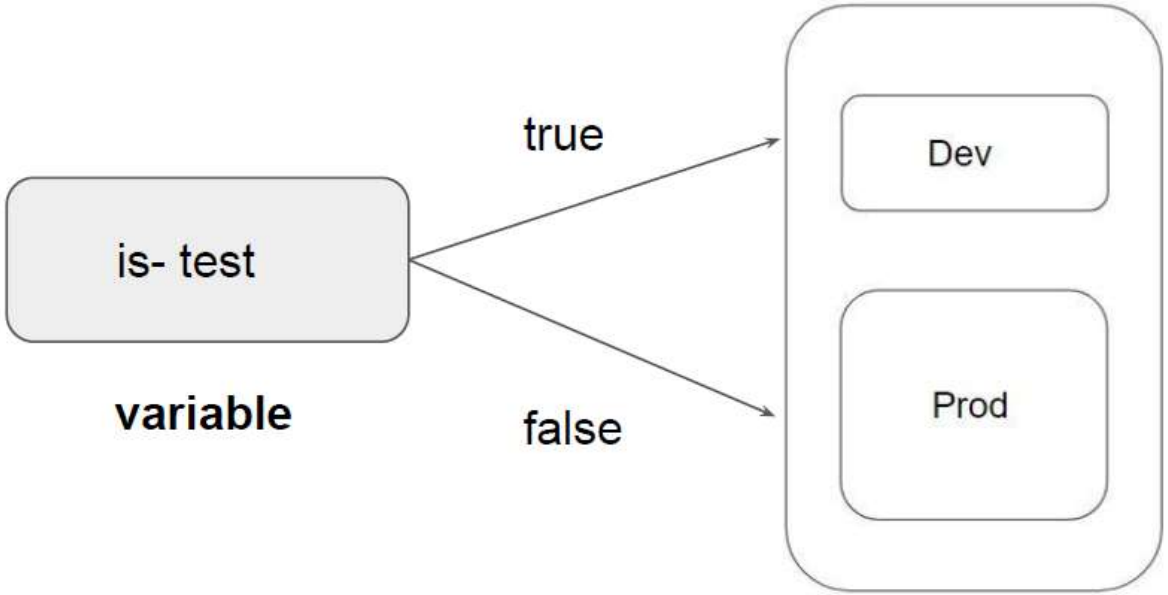
```
condition ? true_val : false_val
```

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

Example

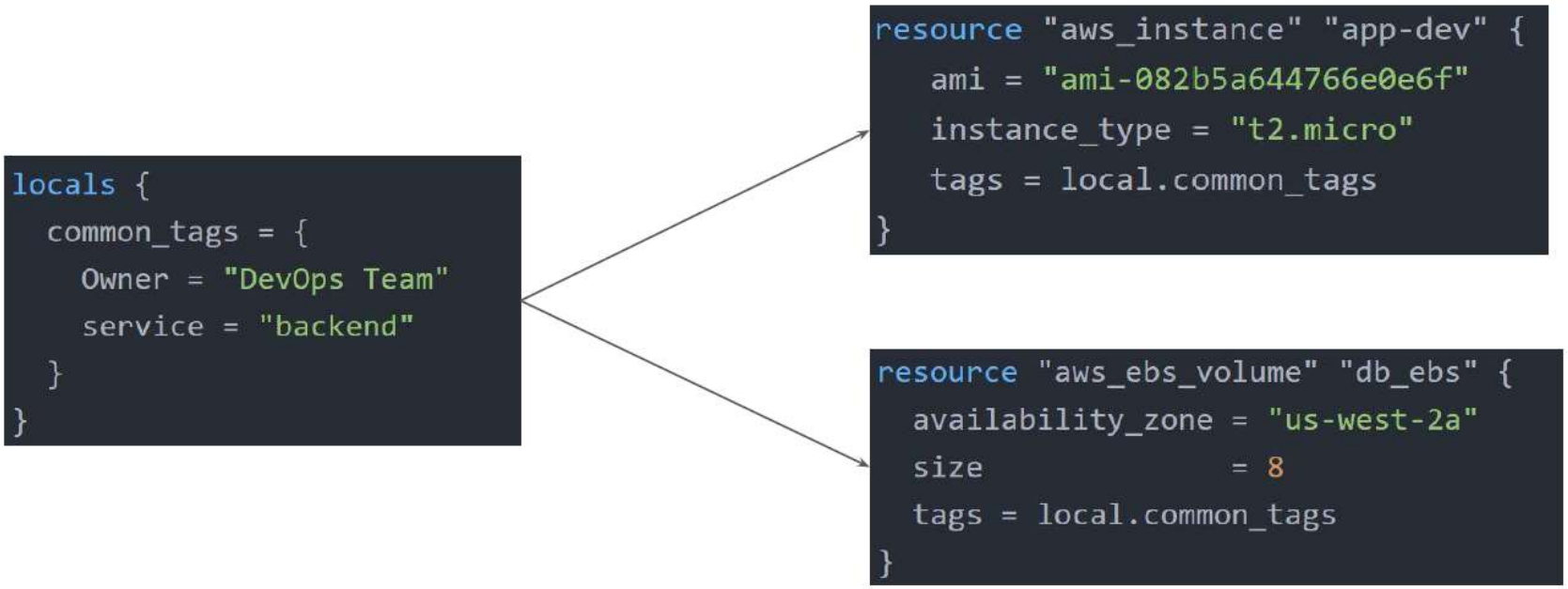
Let's assume that there are two resource blocks as part of terraform configuration.

Depending on the variable value, one of the resource blocks will run.

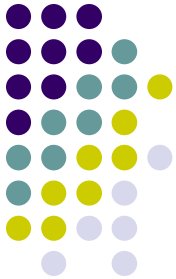


Local values

A local value assigns a name to an expression, allowing it to be used multiple times within a module without repeating it.



Local values expression support



Local Values can be used for multiple different use-cases like having a conditional expression.

```
locals {
  name_prefix = "${var.name != "" ? var.name : var.default}"
}
```

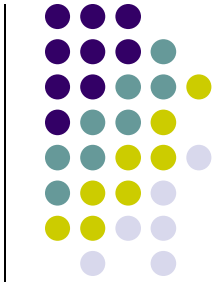
Local values points to note

Local values can be helpful to avoid repeating the same values or expressions multiple times in a configuration.

If overused they can also make a configuration hard to read by future maintainers by hiding the actual values used

Use local values only in moderation, in situations where a single value or result is used in many places and that value is likely to be changed in future.

Terraform function



The Terraform language includes a number of built-in functions that you can use to transform and combine values.

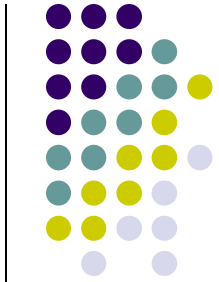
The general syntax for function calls is a function name followed by comma-separated arguments in parentheses:

function (argument1, argument2)

Example:

```
> max(5, 12, 9)
  12
```

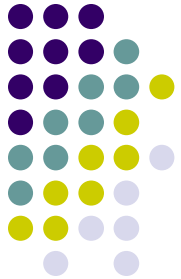
List of available functions



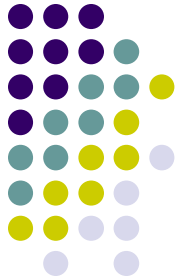
The Terraform language does not support user-defined functions, and so only the functions built in to the language are available for use

- Numeric
- String
- Collection
- Encoding
- Filesystem
- Date and Time
- Hash and Crypto
- IP Network
- Type Conversion

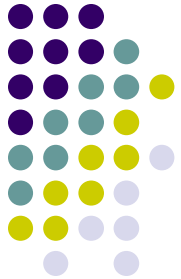
C



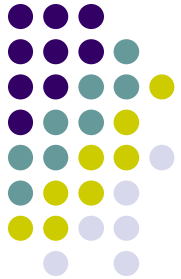
C



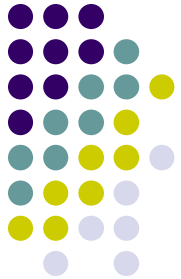
C



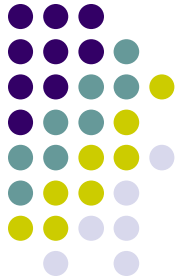
C



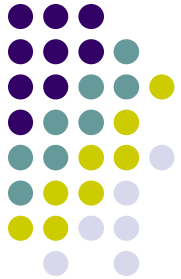
C



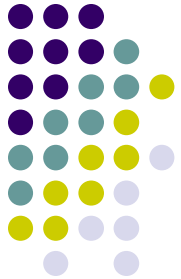
C



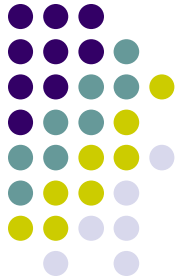
C



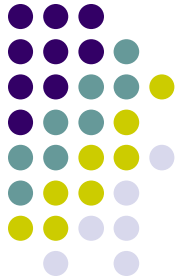
C



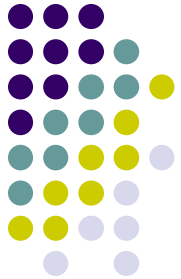
C



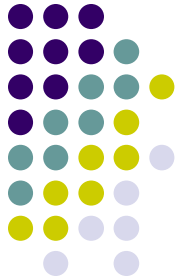
C



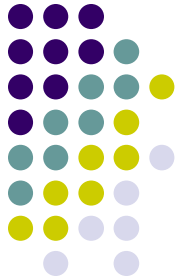
C



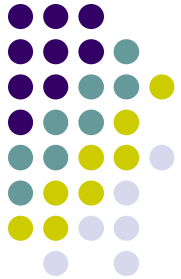
C



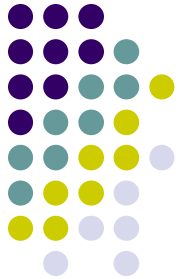
C



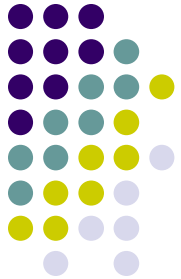
C



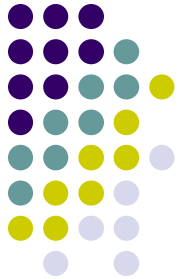
C



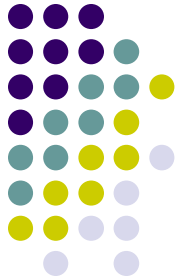
C



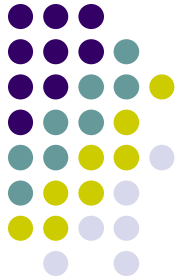
C



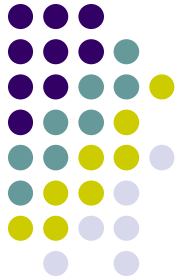
C



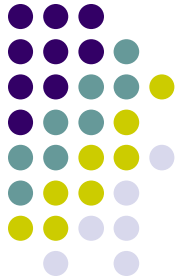
C



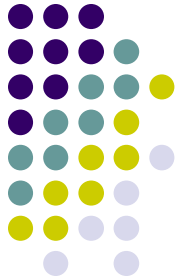
C



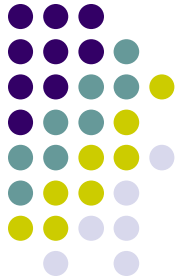
C



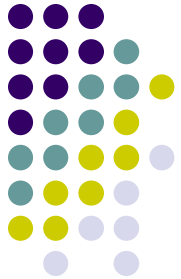
C



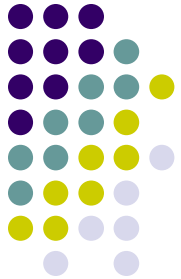
C



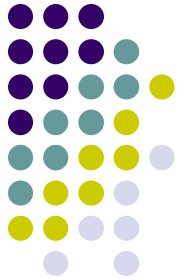
C



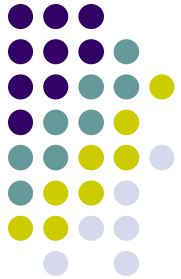
C



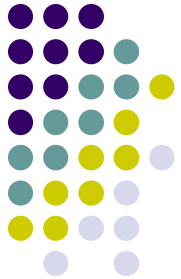
C



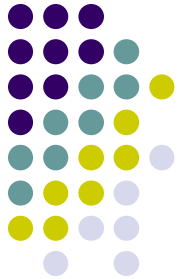
C



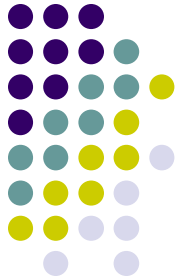
C



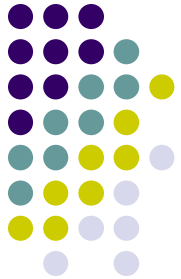
C



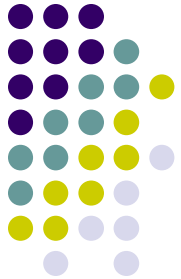
C



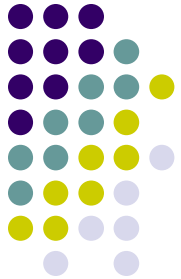
C



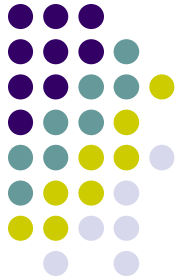
C



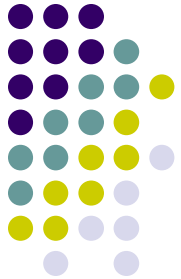
C



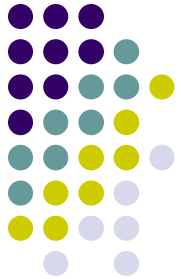
C



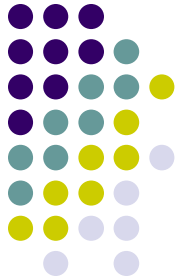
C



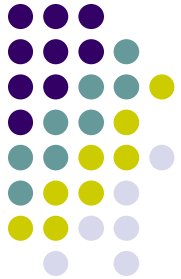
C



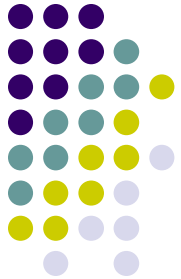
C



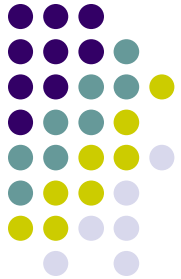
C



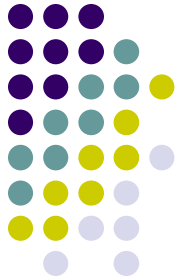
C



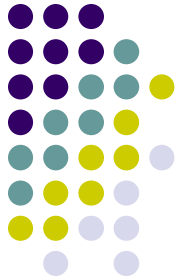
C



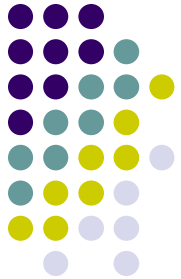
C



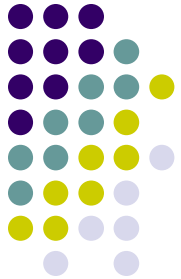
C



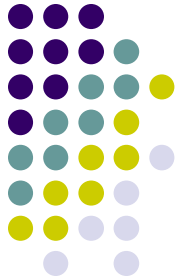
C



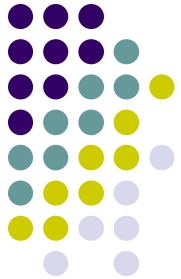
C



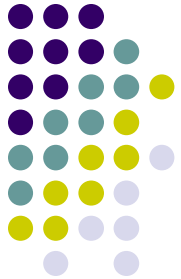
C



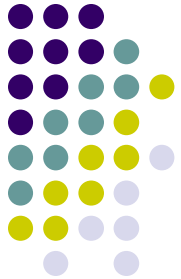
C



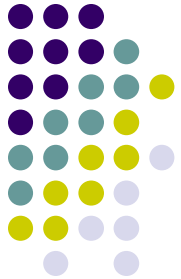
C



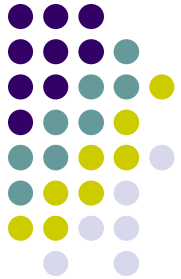
C



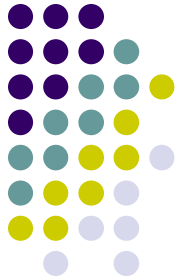
C



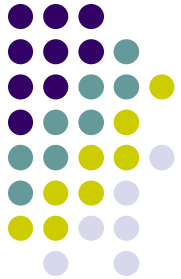
C



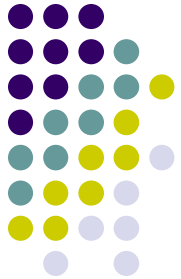
C



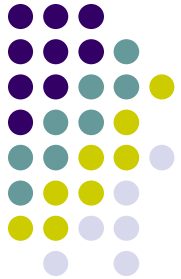
C



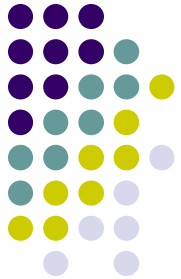
C



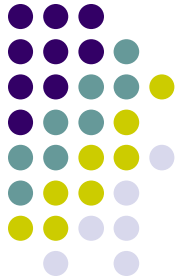
C



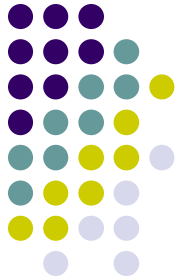
C



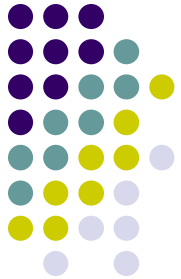
C



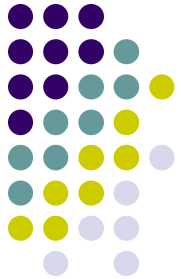
C



C



C



C