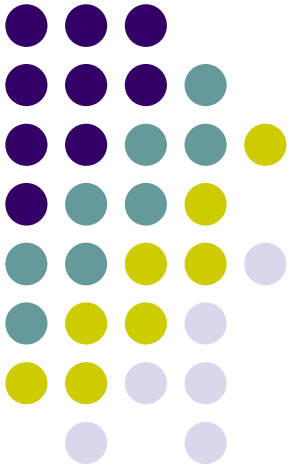
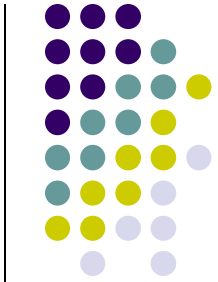
HashiCorp Certified: Terraform Associate

Raj



Configuration Management Vs Infrastructure Orchestration

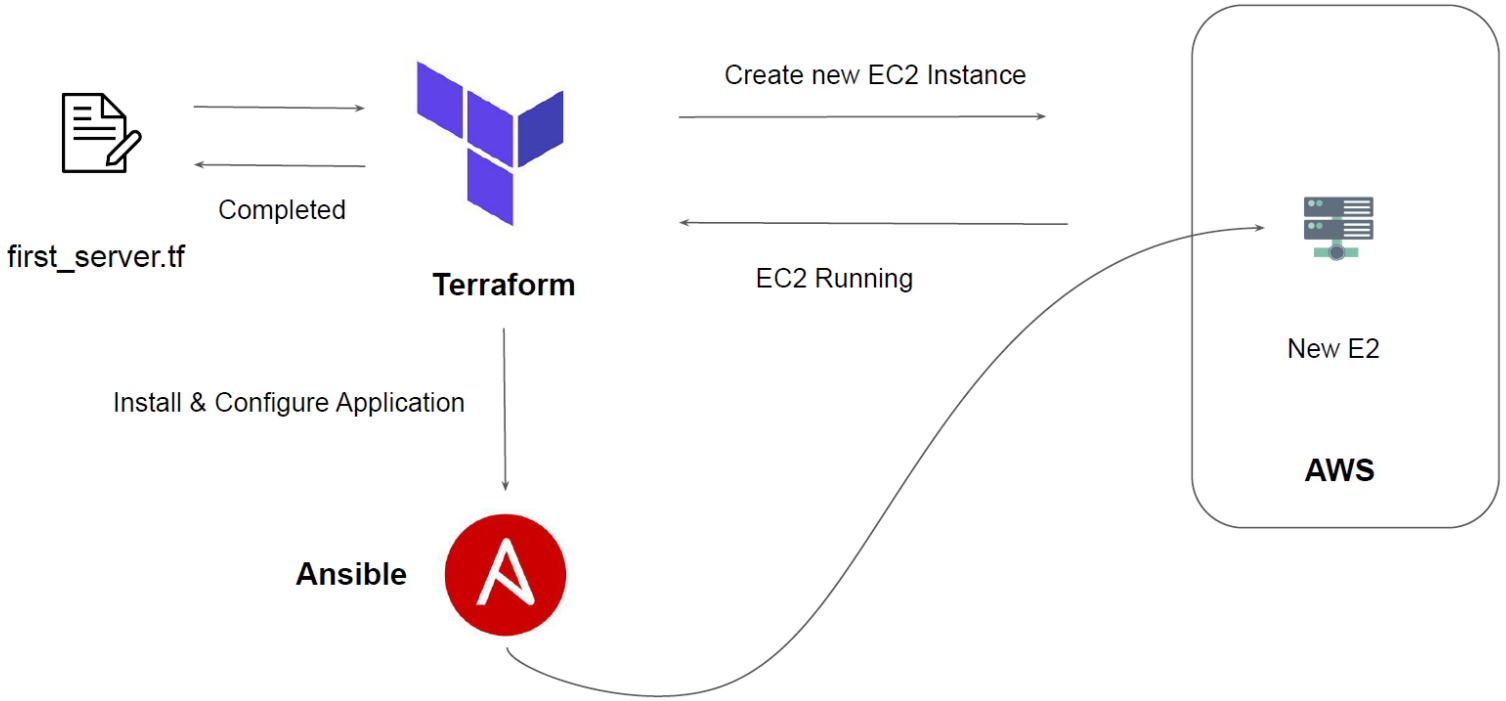


Ansible, Chef, Puppet are configuration management tools which means that they are primarily designed to install and manage software on existing servers.

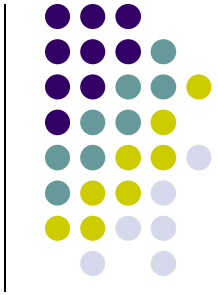
Terraform, CloudFormation are the infrastructure orchestration tools which basically means they can provision the servers and infrastructure by themselves.

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

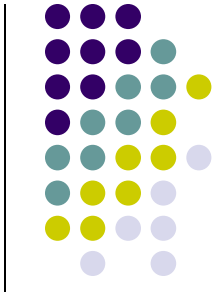
i) Is your infrastructure going to be vendor specific in longer term ? Example AWS.

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.

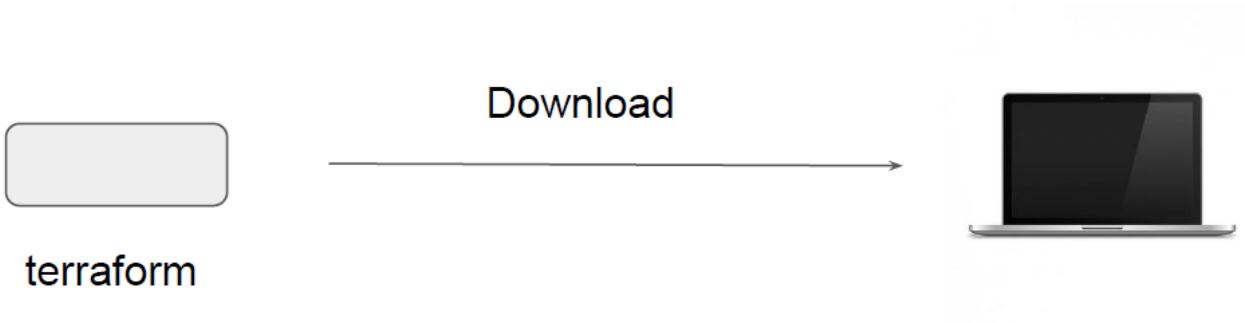
iv) Easily extensible with the help of plugins.

v) Free !!!

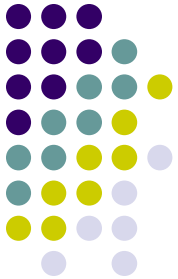
Terraform Installation

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

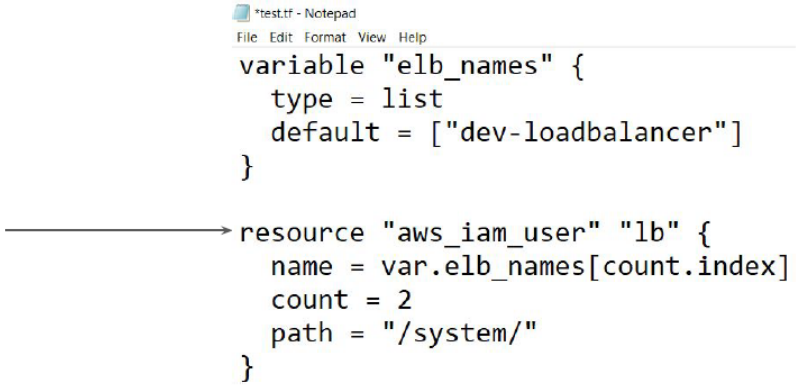
Terraform editor

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

Downsides:

- Slower Development
- Limited Features

HashiCorp
Terraform

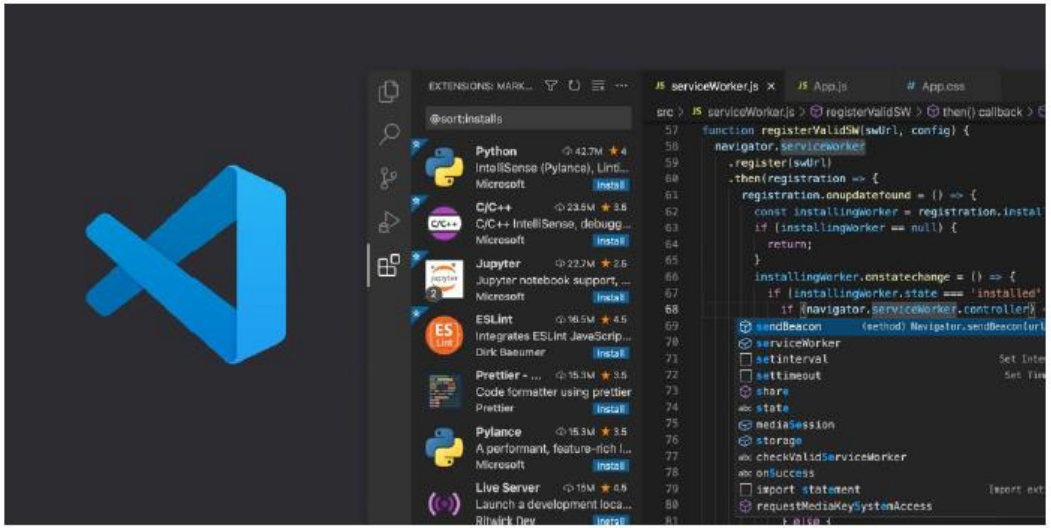


Terraform Editor

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

Advantages:

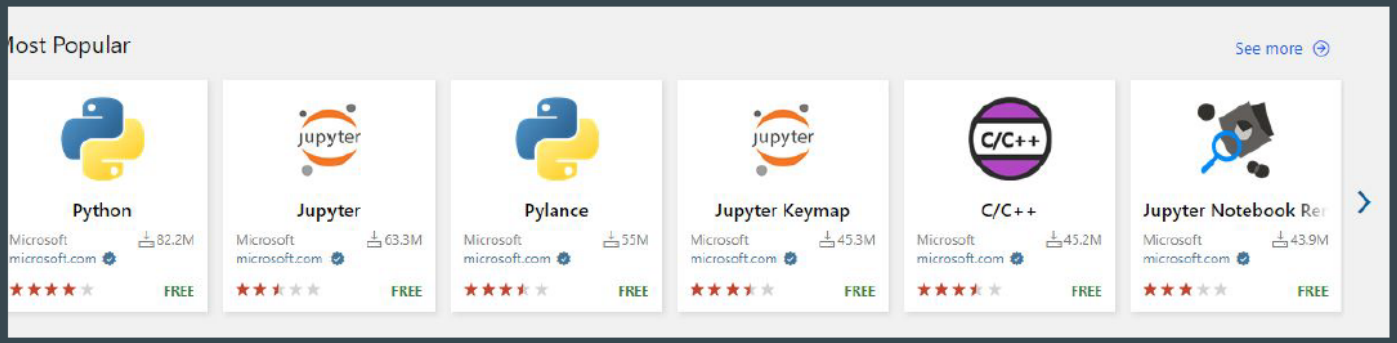
1. Supports Windows, Mac, Linux
2. Supports Wide variety of programming languages.
3. Many Extensions.



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.

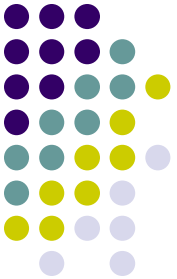
demo.tf

```
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

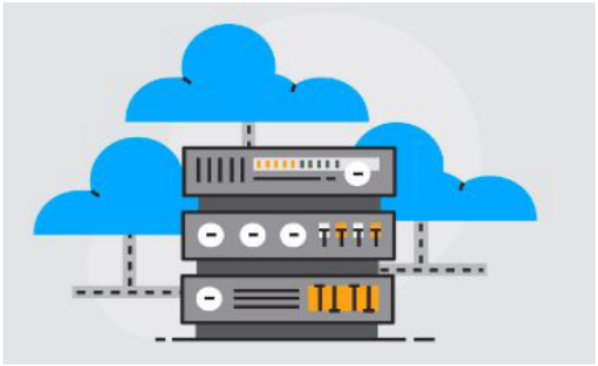


Let’s start Rolling !

Let’s start

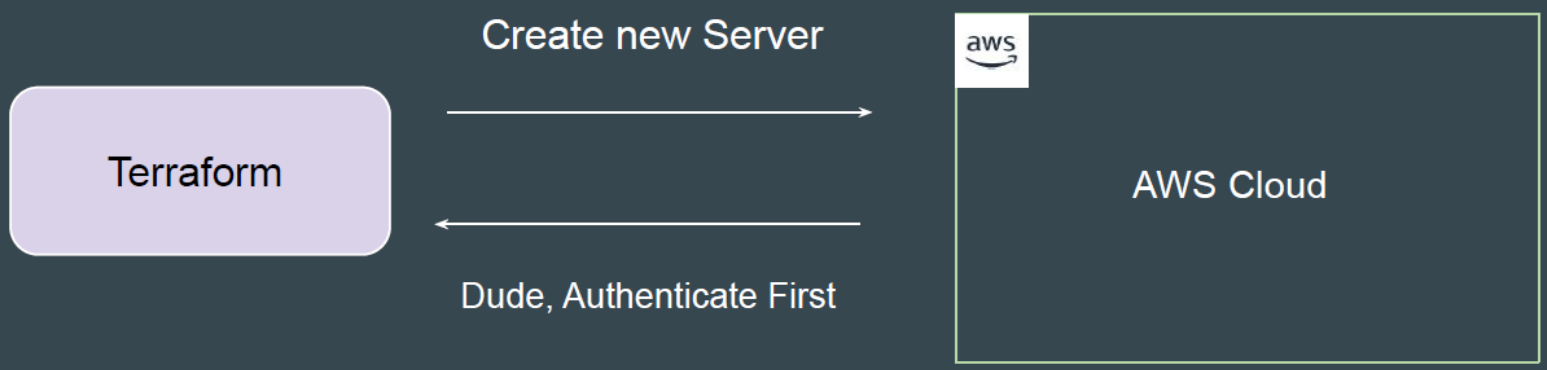
i) Create a new AWS Account.

ii) Begin the course

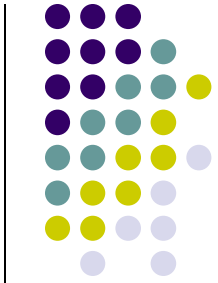


Authentication & Authorization

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.

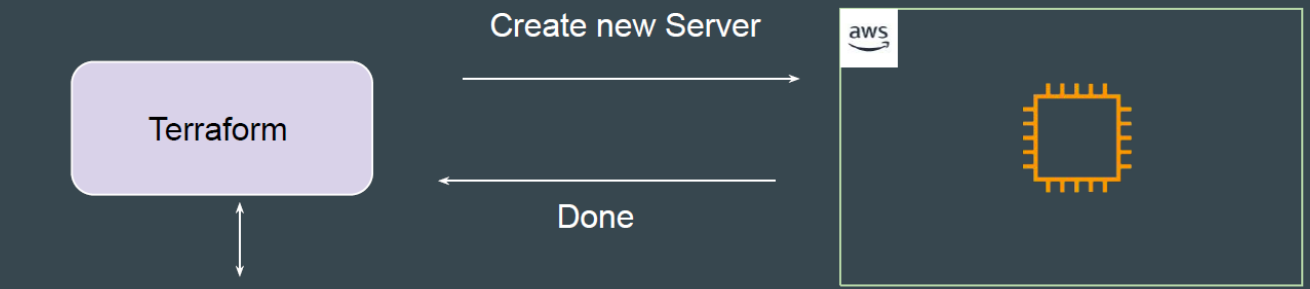
Authorization is the process of verifying what they have access to

Example:

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

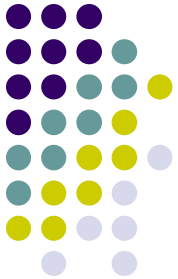
Terraform use case

Terraform needs access credentials with relevant permissions to create and manage the environments.



username	password
Bob	pwd928#

Access credentials



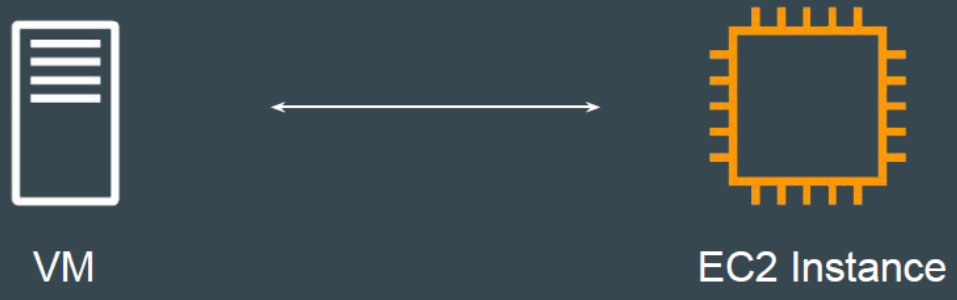
Depending on the provider, the type of access credentials would change.

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.

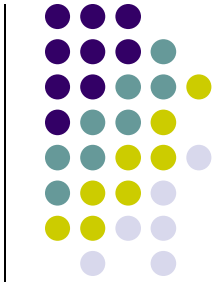


Available Regions

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.

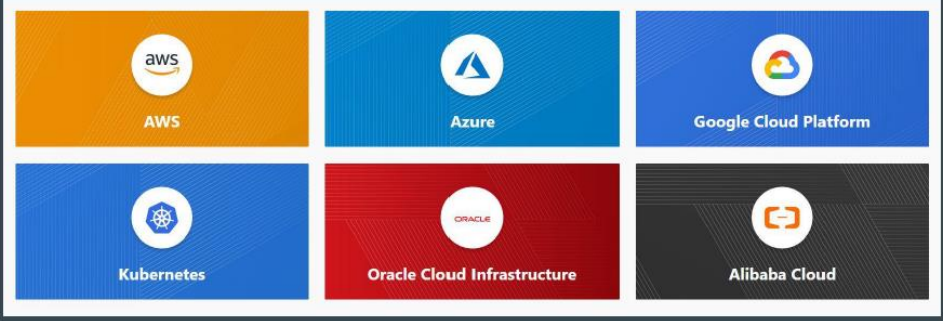
- CPU
- Memory
- Storage
- Operating System

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

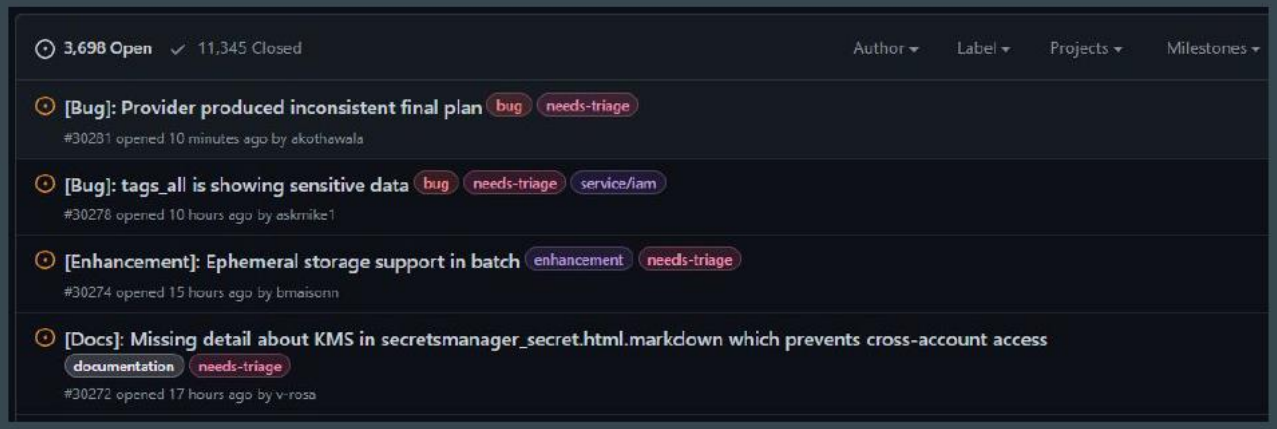
The core concepts, standard syntax remains similar across all providers.

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

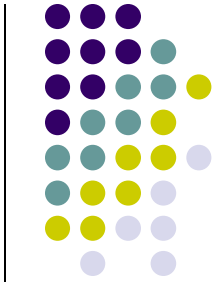
Reporting bug with Provider

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.



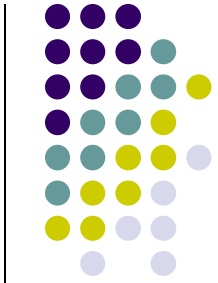
Provider Maintainers



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.

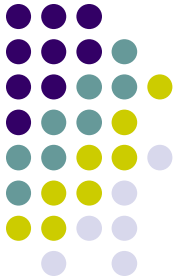
Provider Namespace



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

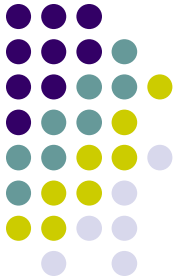
```
provider "aws" {
  region     = "us-west-2"
  access_key = "PUT-YOUR-ACCESS-KEY-HERE"
  secret_key = "PUT-YOUR-SECRET-KEY-HERE"
}
```

HashiCorp Maintained

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

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

Non-HashiCorp Maintained

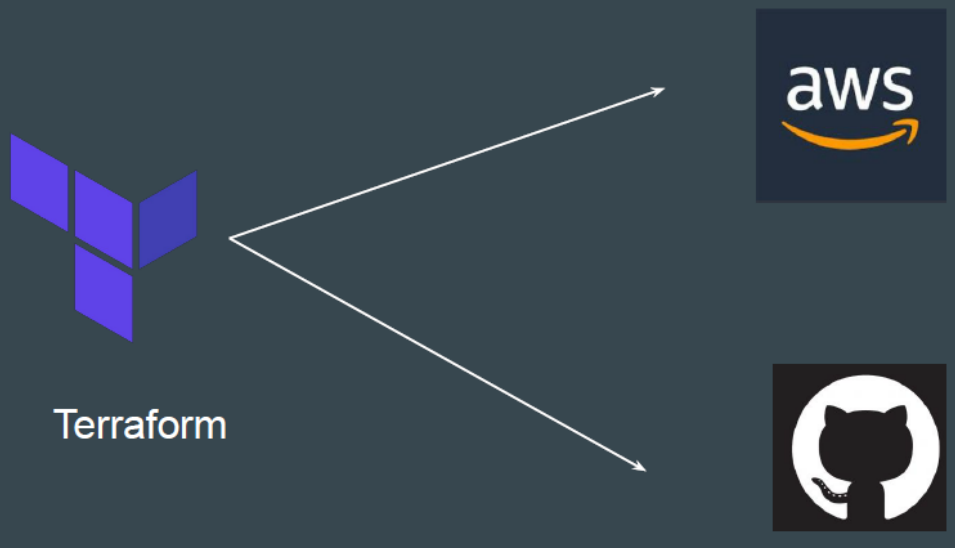
Day 2



Destroy

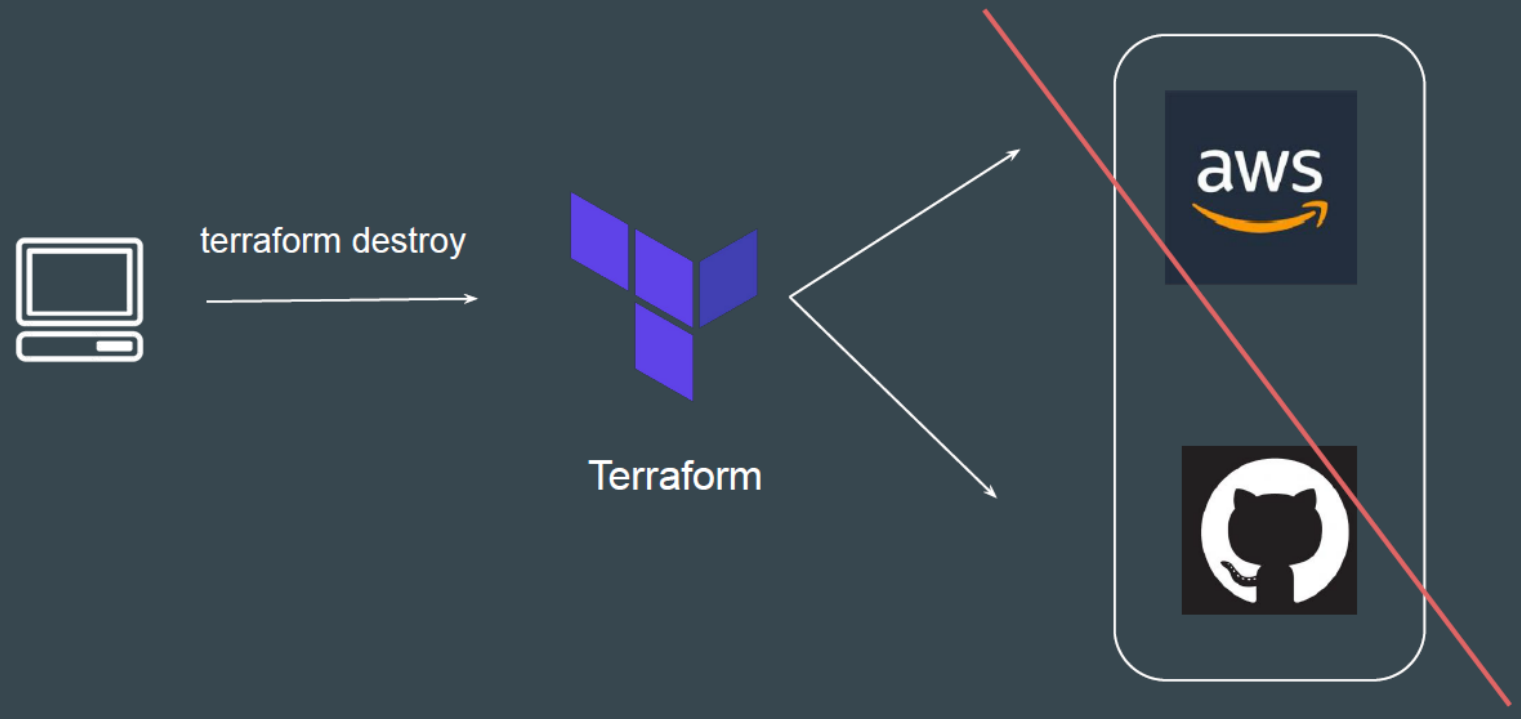
If you keep the infrastructure running, you will get charged for it.

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



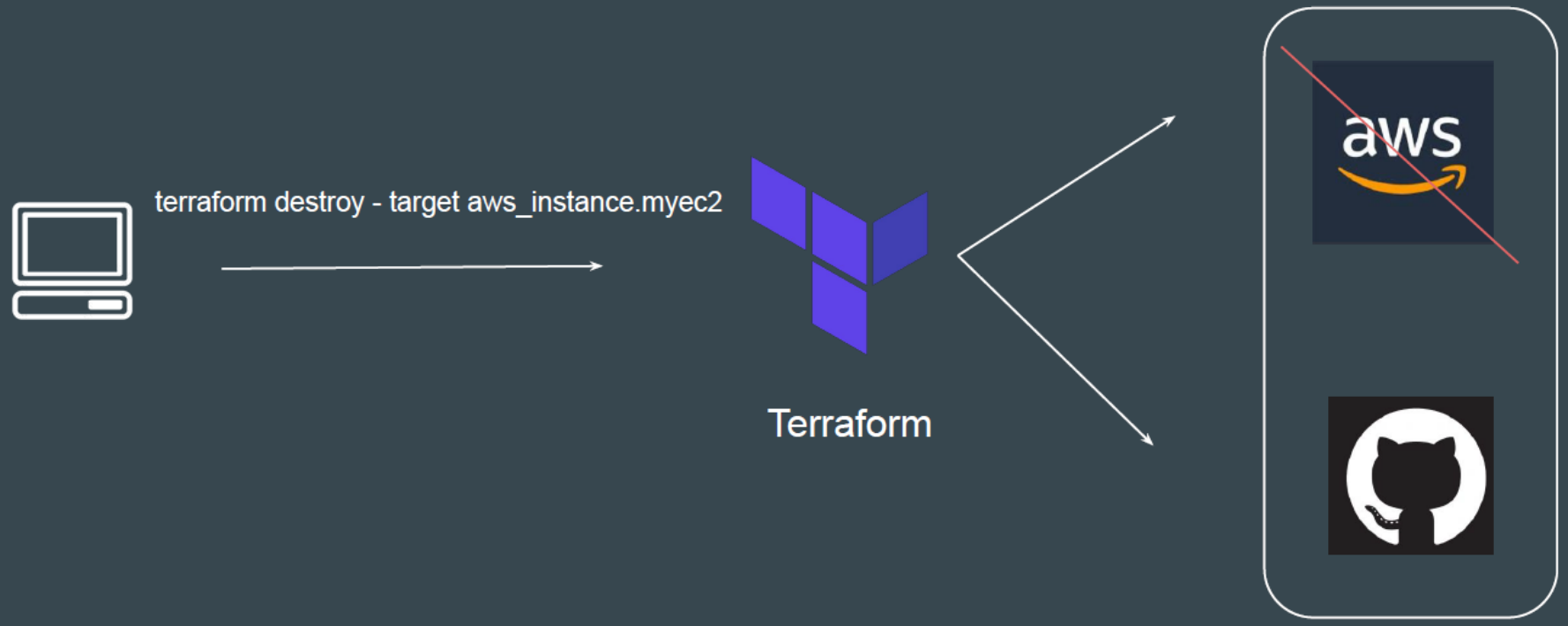
Destroy all

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

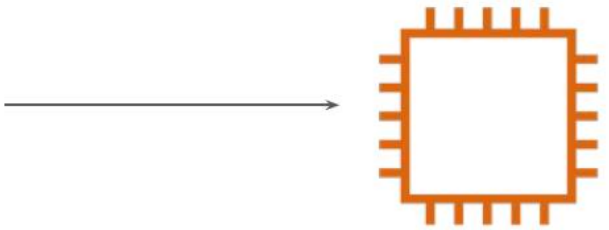
```
resource "aws_instance" "myec2" {
    ami = "ami-00c39f71452c08778"
    instance_type = "t2.micro"
}
```

```
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"
}
```

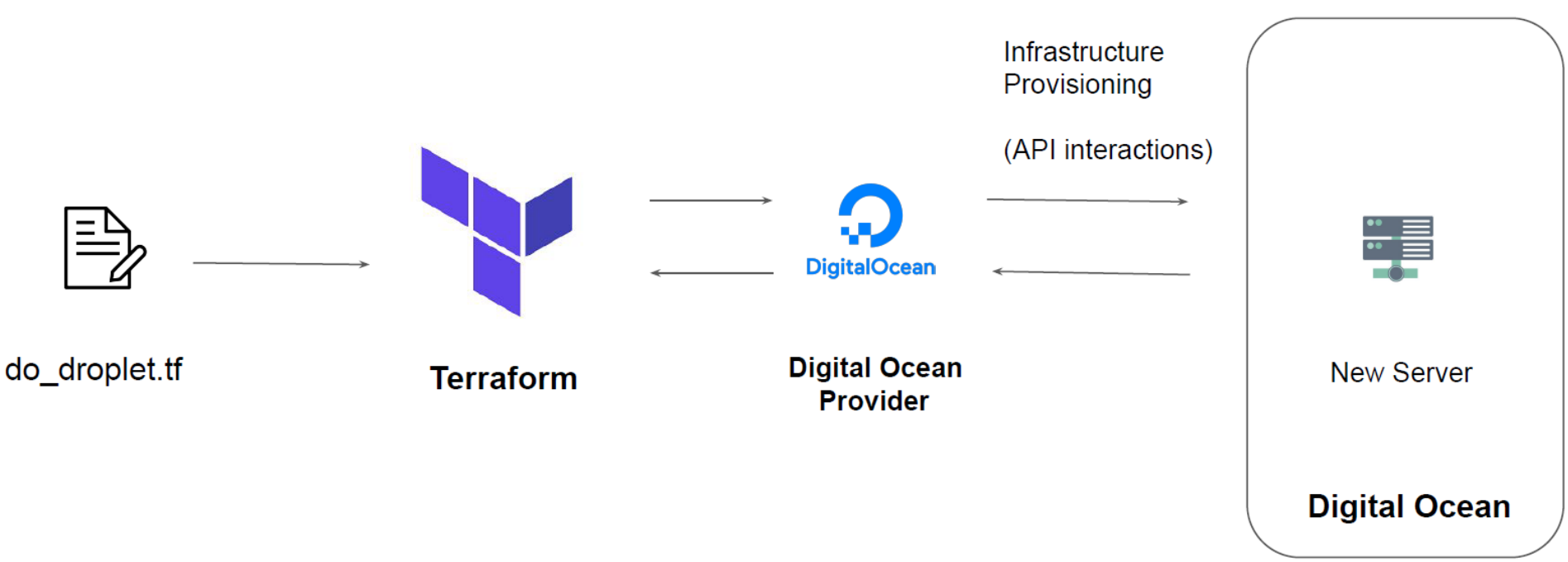
t2.medium

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.

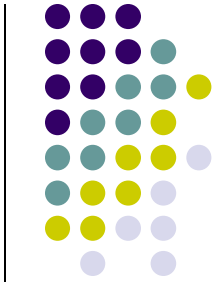
.

DigitalOcean

Version 1

Version 2

Explicit provider versioning

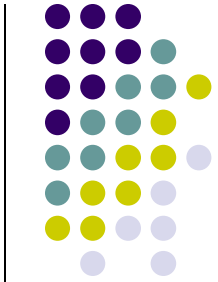


During terraform init, if version argument is not specified, the most recent provider will be downloaded during initialization.

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.

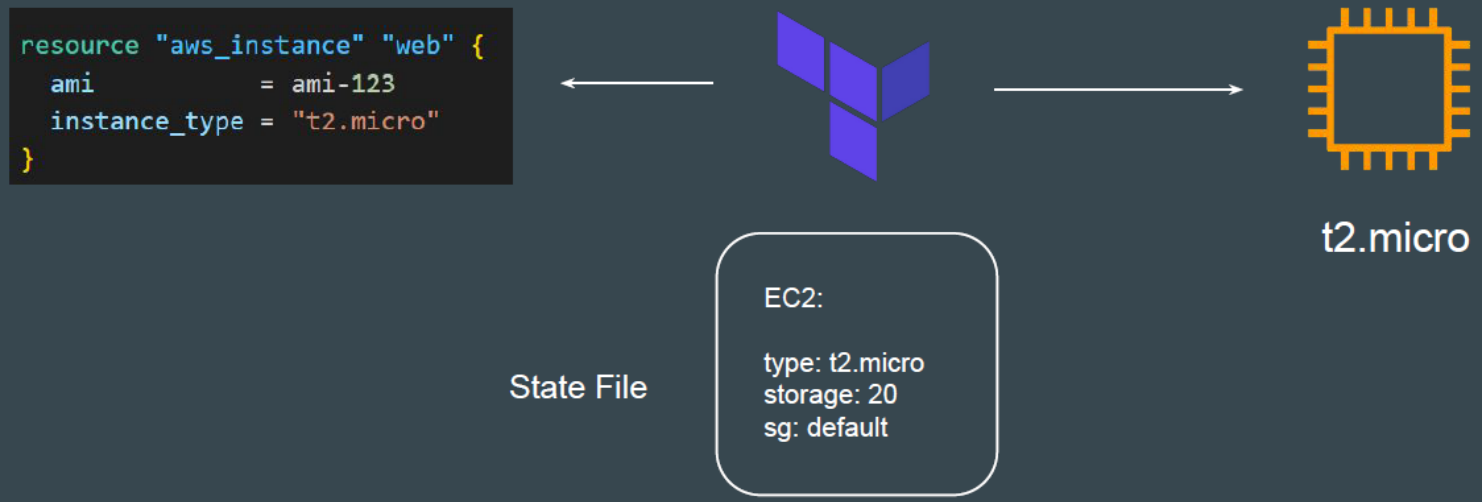
You can override that behavior by adding the -upgrade option when you run terraform init,

```
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

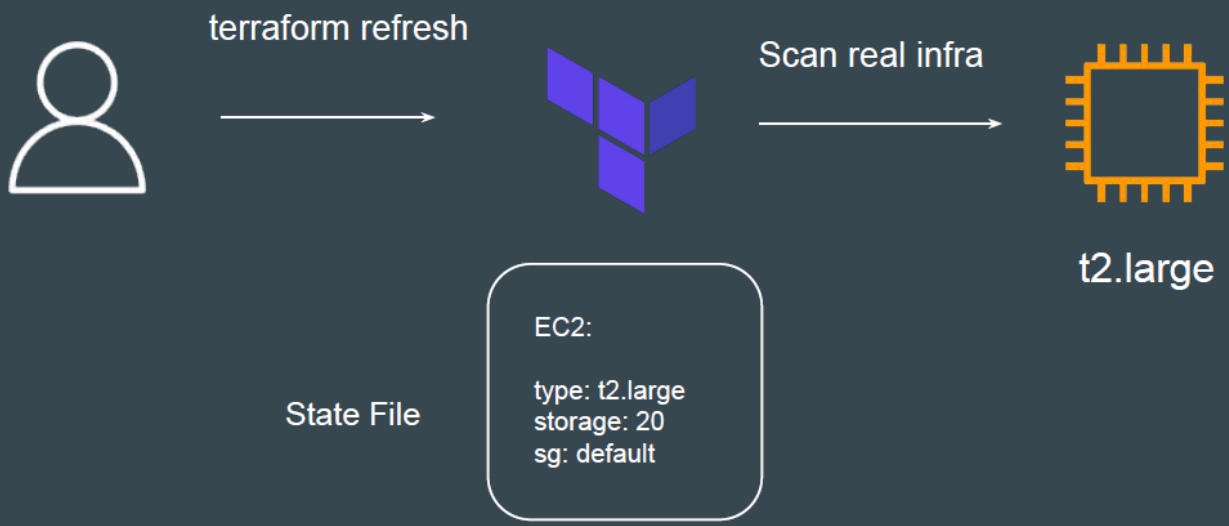
Terraform can create an infrastructure based on configuration you specified.

It can happen that the infrastructure gets modified manually.

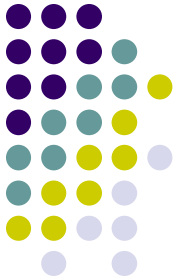


Challenge

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

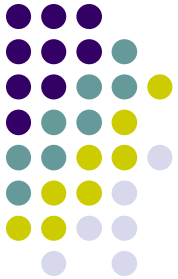


Challenge



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.

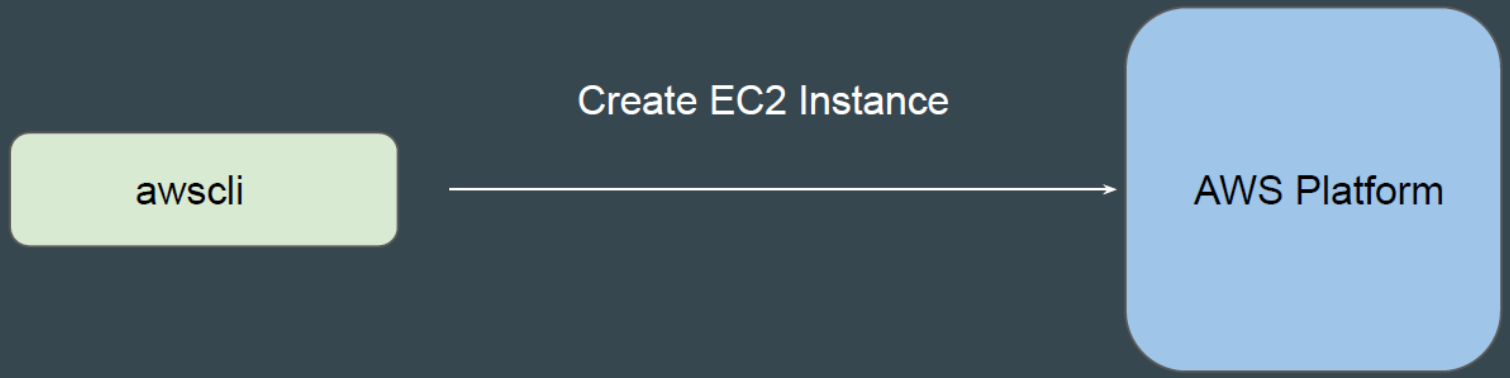
aws-provider-config.tf > ...

```
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?]
  }
```

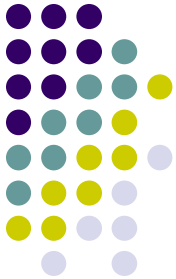
Attributes

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

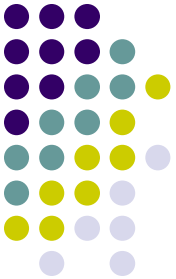
Attribute	Value
public_ip	52.72.52.72

```
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

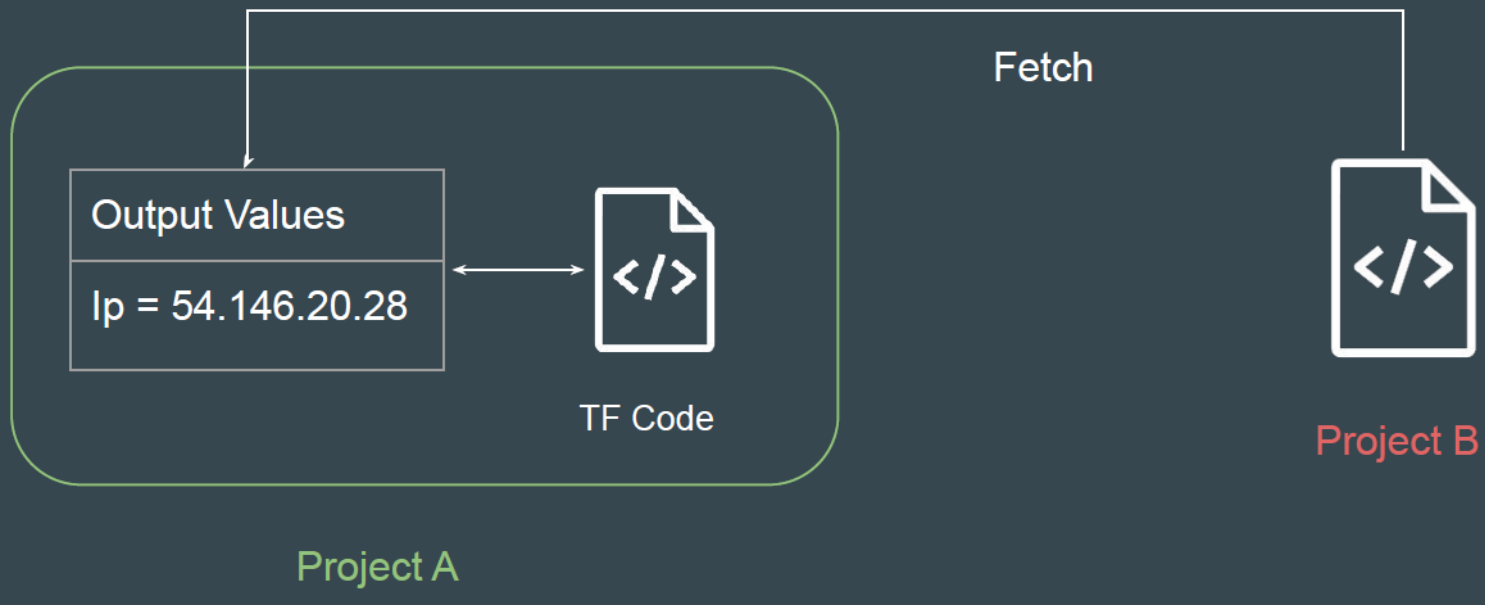


Create a Elastic IP (Public IP) resource in AWS and output the value of the EIP.

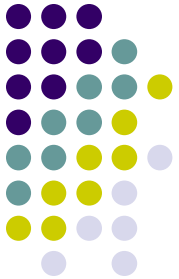
```
Plan: 1 to add, 0 to change, 0 to destroy.

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.

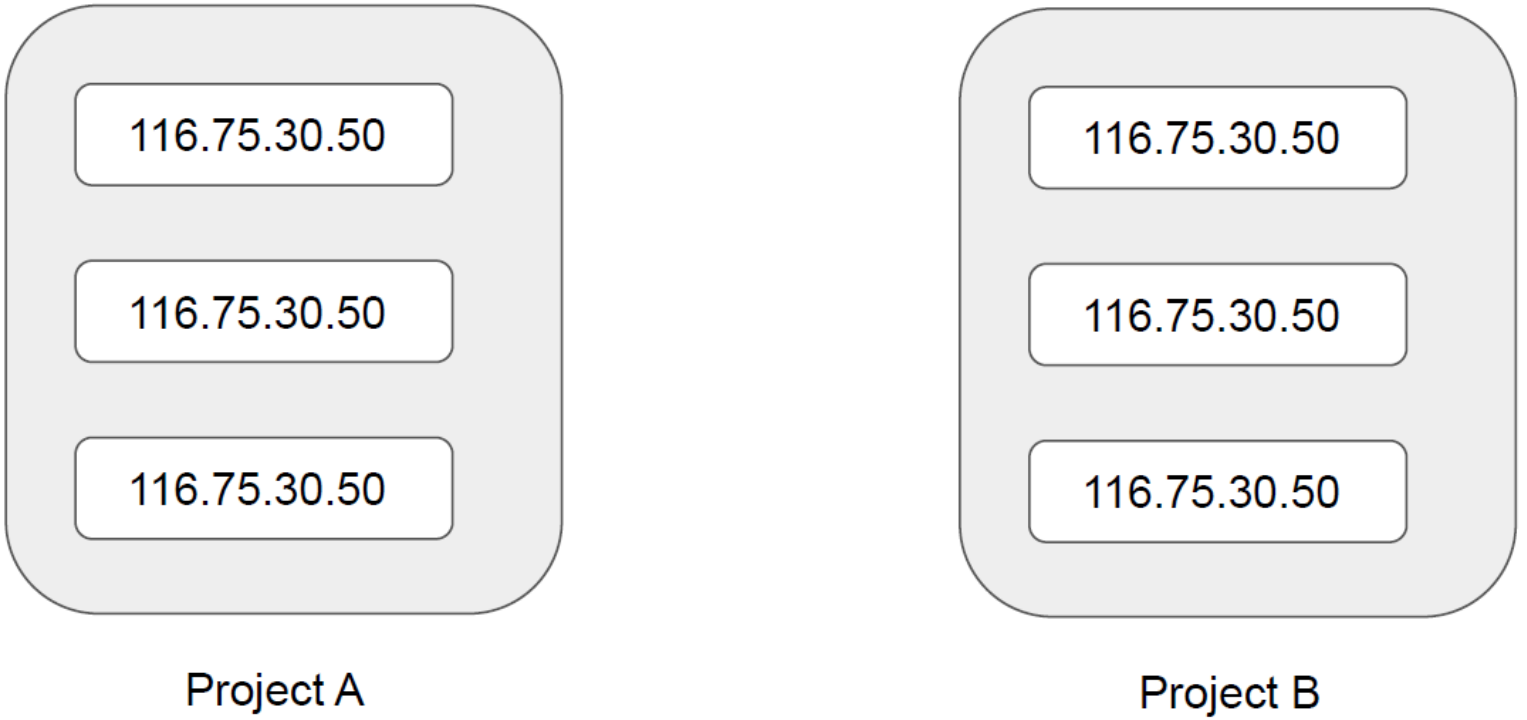


Day3



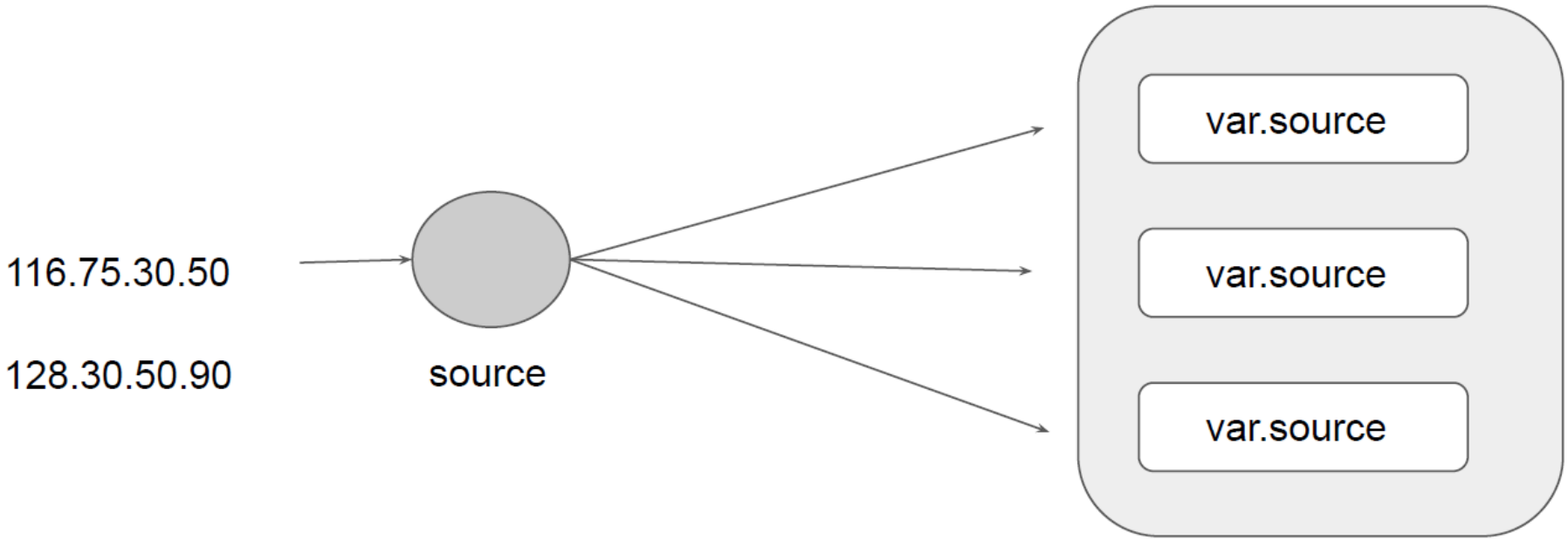
Static=work

Repeated static values can create more work in the future.

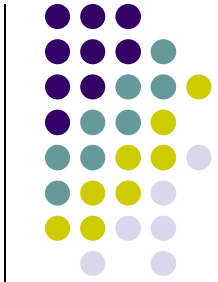


Value – Central location

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



Variables assignment

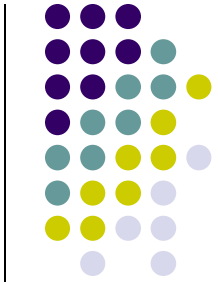


Variables in Terraform can be assigned values in multiple ways.

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.

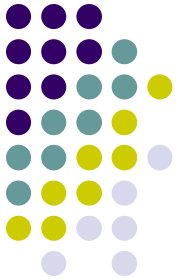
Variable

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"
}
```

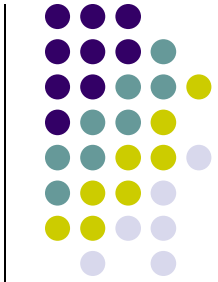
```
resource "aws_instance" "instance-2" {
   ami = "ami-082b5a644766e0e6f"
   instance_type = "t2.micro"
}
```

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

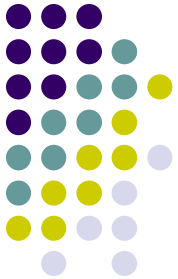


In resource blocks where count is set, an additional count object is available in expressions, so you can modify the configuration of each instance.

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

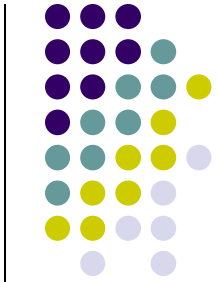
Having a username like loadbalancer0, loadbalancer1 might not always be suitable.

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:

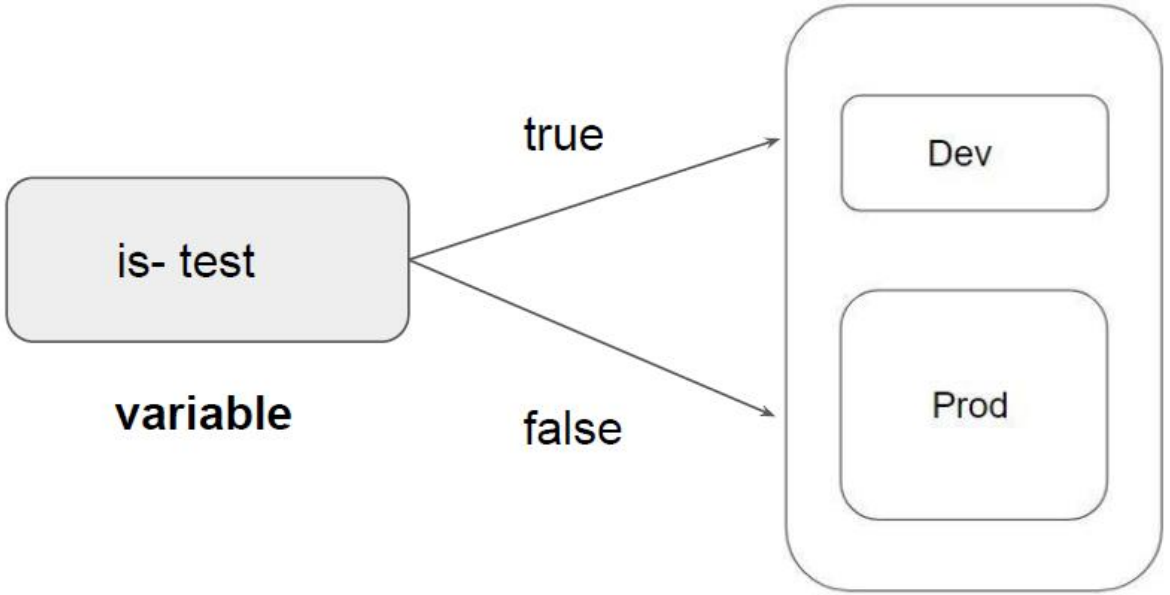
```
condition ? true_val : false_val
```

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

Example

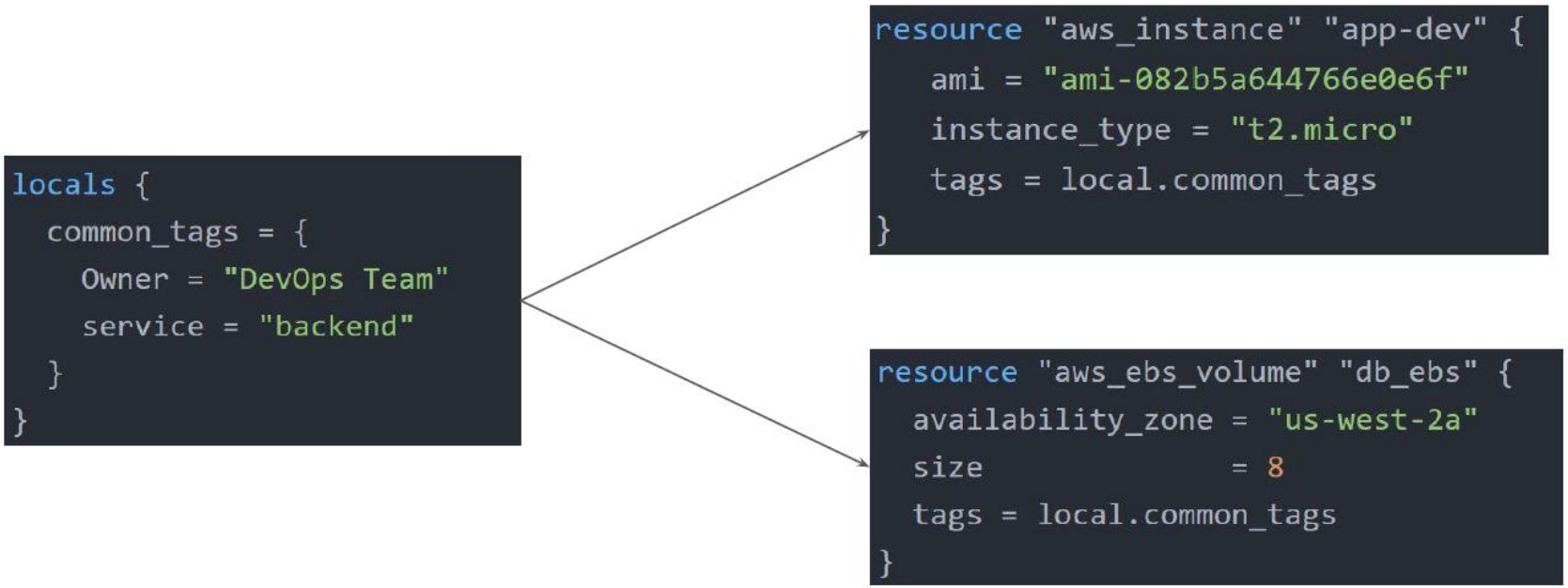
Let's assume that there are two resource blocks as part of terraform configuration.

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

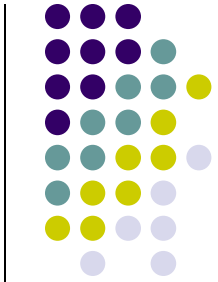


Local values

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}"
}
```

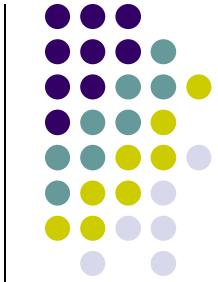
Local values points to note

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.

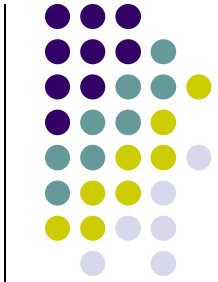
The general syntax for function calls is a function name followed by comma-separated arguments in parentheses:

function (argument1, argument2)

Example:

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

List of available functions

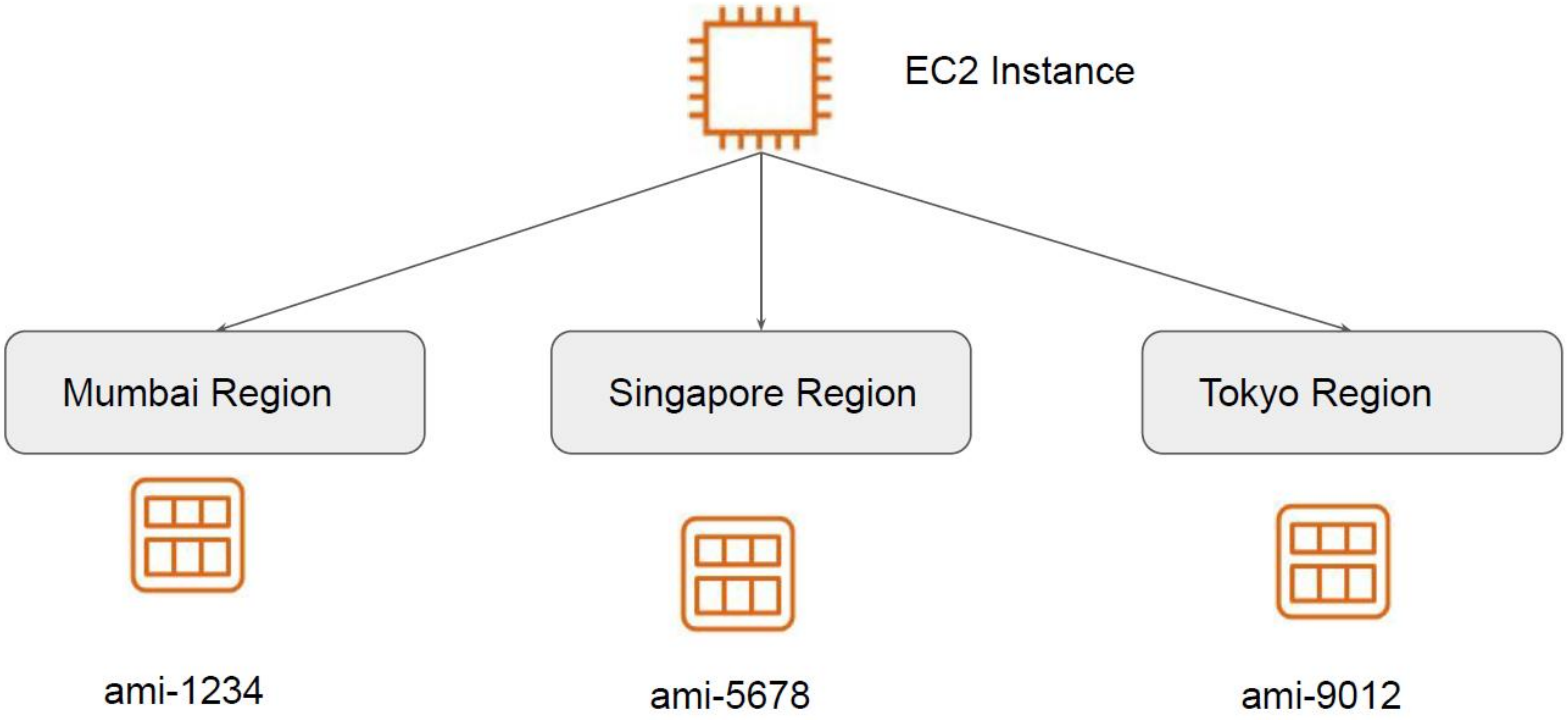


The Terraform language does not support user-defined functions, and so only the functions built in to the language are available for use

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

Data source

Data sources allow data to be fetched or computed for use elsewhere in Terraform configuration.



Data source in action

- Defined under the data block.
- Reads from a specific data source (aws_ami) and exports results under “app_ami”

```
data "aws_ami" "app_ami" {
  most_recent = true
  owners = ["amazon"]


  filter {
    name   = "name"
    values = ["amzn2-ami-hvm*"]
  }
}
```

```
resource "aws_instance" "instance-1" {
    ami = data.aws_ami.app_ami.id
   instance_type = "t2.micro"
}
```

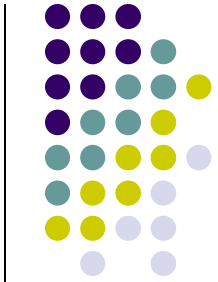
Debugging Terraform

Terraform has detailed logs which can be enabled by setting the TF_LOG environment variable to any value.

You can set TF_LOG to one of the log levels TRACE, DEBUG, INFO, WARN or ERROR to change the verbosity of the logs

```
bash-4.2# terraform plan
2020/04/22 13:45:31 [INFO] Terraform version: 0.12.24
2020/04/22 13:45:31 [INFO] Go runtime version: go1.12.13
2020/04/22 13:45:31 [INFO] CLI args: []string{"/usr/bin/terraform", "plan"}
2020/04/22 13:45:31 [DEBUG] Attempting to open CLI config file: /root/.terraformrc
2020/04/22 13:45:31 [DEBUG] File doesn't exist, but doesn't need to. Ignoring.
2020/04/22 13:45:31 [DEBUG] checking for credentials in "/root/.terraform.d/plugins"
2020/04/22 13:45:31 [INFO] CLI command args: []string{"plan"}
2020/04/22 13:45:31 [TRACE] Meta.Backend: built configuration for "s3" backend with hash value 789489680
2020/04/22 13:45:31 [TRACE] Meta.Backend: backend has not previously been initialized in this working directory
2020/04/22 13:45:31 [DEBUG] New state was assigned lineage "a10f92bf-686d-e6cf-3e9d-755be5c8a6a3"
2020/04/22 13:45:31 [TRACE] Meta.Backend: moving from default local state only to "s3" backend
```

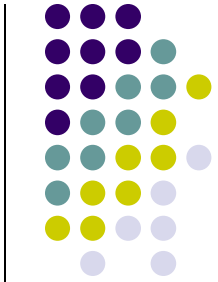
Terraform Debug



TRACE is the most verbose and it is the default if TF_LOG is set to something other than a log level name.

To persist logged output you can set TF_LOG_PATH in order to force the log to always be appended to a specific file when logging is enabled.

Readability



Anyone who is into programming knows the importance of formatting the code for readability.

The terraform fmt command is used to rewrite Terraform configuration files to take care of the overall formatting.

```
provider "aws" {
      region     = "us-west-2"
  access_key = "AKIAQIW66DN2W7WOYRGY"
  secret_key     = "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"
  version    = ">=2.10,<=2.30"
}
```

Format

Before fmt

```
provider "aws" {
      region     = "us-west-2"
  access_key = "AKIAQIW66DN2W7WOYRGY"
  secret_key    = "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"
  version    = ">=2.10,<=2.30"
}
```

After fmt

```
provider "aws" {
  region     = "us-west-2"
  access_key = "AKIAQIW66DN2W7WOYRGY"
  secret_key = "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"
  version    = ">=2.10,<=2.30"
}
```

Terraform validate

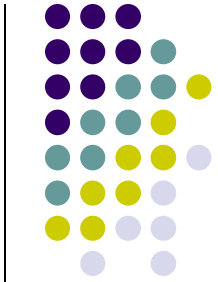
Terraform Validate primarily checks whether a configuration is syntactically valid.

It can check various aspects including unsupported arguments, undeclared variables and others.

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

```
bash-4.2# terraform validate

Error: Unsupported argument

  on validate.tf line 10, in resource "aws_instance" "myec2":
  10:   sky = "blue"

An argument named "sky" is not expected here.
```

Terraform load order



Terraform generally loads all the configuration files within the directory specified in alphabetical order.

The files loaded must end in either .tf or .tf.json to specify the format that is in use.

Dynamic block

Dynamic Block allows us to dynamically construct repeatable nested blocks which is supported inside resource, data, provider, and provisioner blocks:

```
dynamic "ingress" {
  for_each = var.ingress_ports
  content {
    from_port   = ingress.value
    to_port     = ingress.value
    protocol    = "tcp"
    cidr_blocks = ["0.0.0.0/0"]
  }
}
```

Iterator

The iterator argument (optional) sets the name of a temporary variable that represents the current element of the complex value

If omitted, the name of the variable defaults to the label of the dynamic block ("ingress" in the example above).

```
dynamic "ingress" {
  for_each = var.ingress_ports
  content {
    from_port   = ingress.value
    to_port     = ingress.value
    protocol    = "tcp"
    cidr_blocks = ["0.0.0.0/0"]
  }
}
```

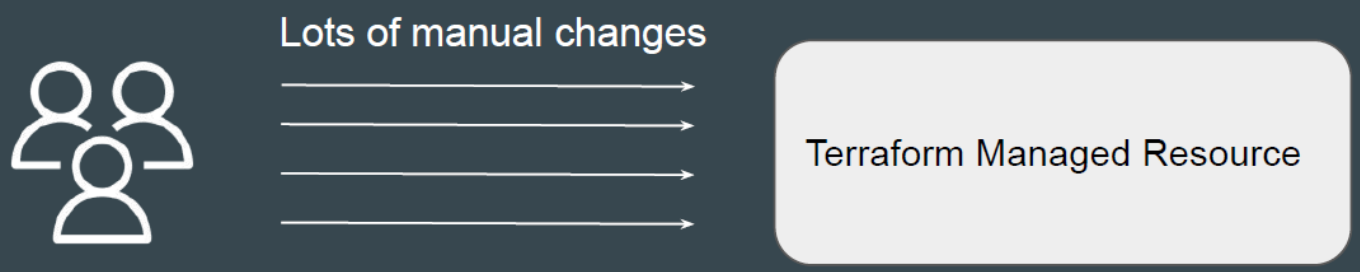
```
dynamic "ingress" {
  for_each = var.ingress_ports
  iterator = port
  content {
    from_port   = port.value
    to_port     = port.value
    protocol    = "tcp"
    cidr_blocks = ["0.0.0.0/0"]
  }
}
```

Use case of manual change

You have created a new resource via Terraform.

Users have made a lot of manual changes (both infrastructure and inside the server)

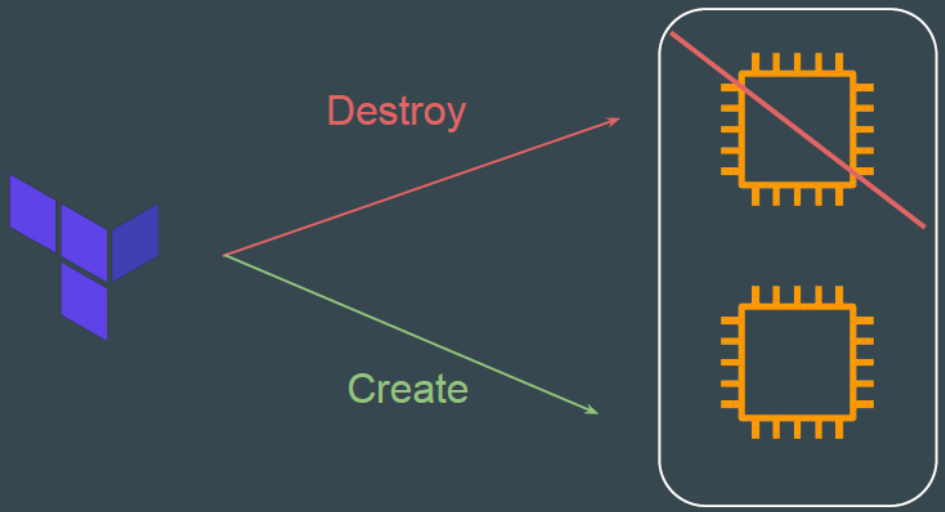
Two ways to deal with this: Import Changes to Terraform / Delete & Recreate the resource



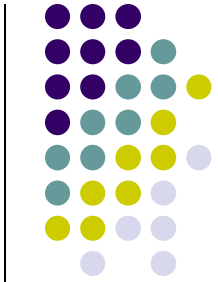
Recreating the resource

The -replace option with terraform apply to force Terraform to replace an object even though there are no configuration changes that would require it.

terraform apply -replace="aws_instance.web"



Terraform replace vs taint



Similar kind of functionality was achieved using terraform taint command in older versions of Terraform.

For Terraform v0.15.2 and later, HashiCorp recommend using the -replace option with terraform apply

Splat expression

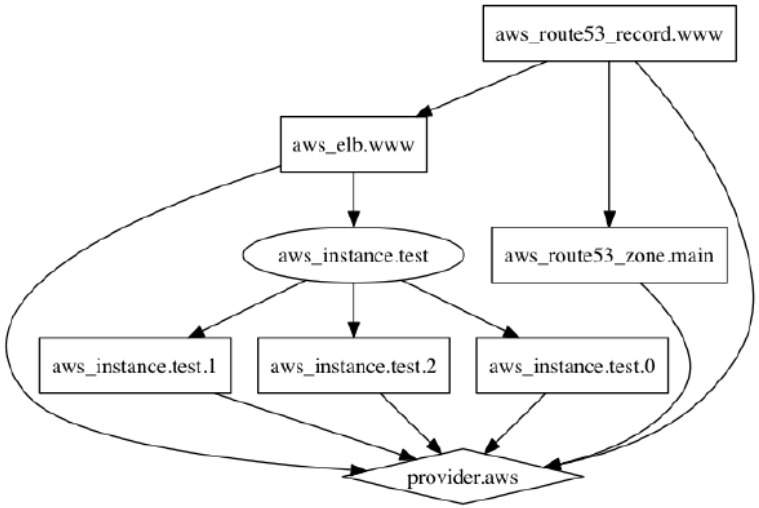
Splat Expression allows us to get a list of all the attributes.

```
resource "aws_iam_user" "lb" {
  name = "iamuser.${count.index}"
  count = 3
  path = "/system/"
}

output "arns" {
  value = aws_iam_user.lb[*].arn
}
```

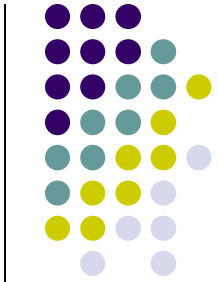
Terraform Graph

The terraform graph command is used to generate a visual representation of either a configuration or execution plan

The output of terraform graph is in the DOT format, which can easily be converted to an image.



Terraform plan to File



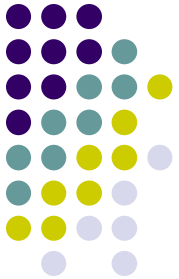
The generated terraform plan can be saved to a specific path.

This plan can then be used with terraform apply to be certain that only the changes shown in this plan are applied.

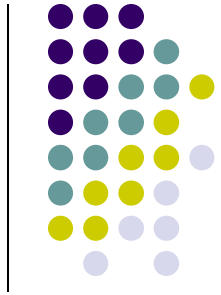
Example:

terraform plan -out=path

Day4



Terraform Provisioners



Till now we have been working only on creation and destruction of infrastructure scenarios.

Let's take an example:

We created a web-server EC2 instance with Terraform.

Problem: It is only an EC2 instance, it does not have any software installed.

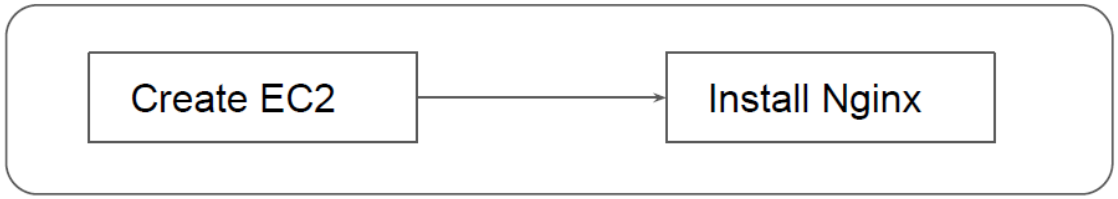
What if we want a complete end to end solution ?

Provisioner need

Provisioners are used to execute scripts on a local or remote machine as part of resource creation or destruction.

Let's take an example:

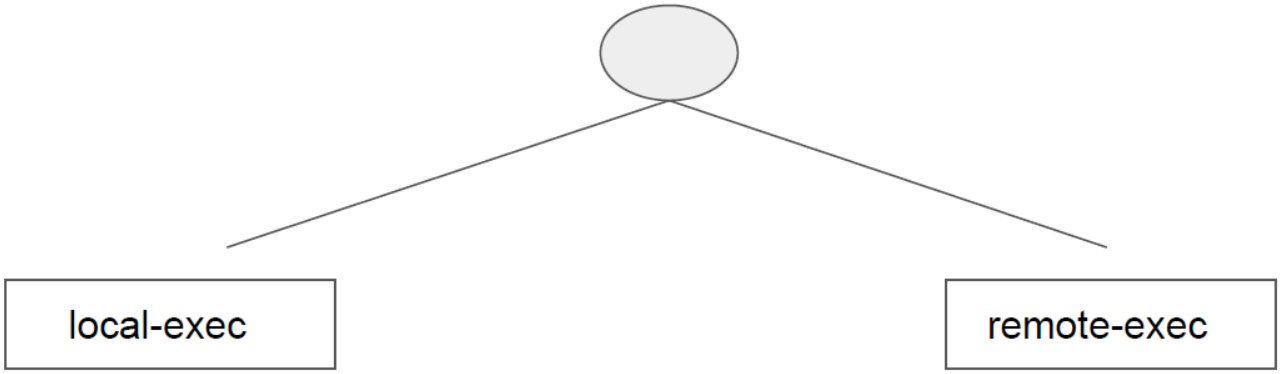
On creation of Web-Server, execute a script which installs Nginx web-server.



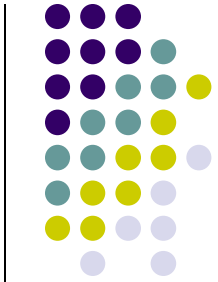
Provisioner types

Terraform has capability to turn provisioners both at the time of resource creation as well as destruction.

There are two main types of provisioners:



Local exec provisioners

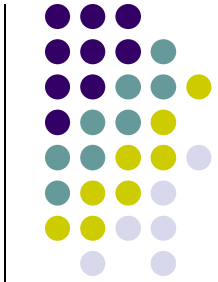


local-exec provisioners allow us to invoke local executable after resource is created

Let's take an example:

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

  provisioner "local-exec" {
    command = "echo ${aws_instance.web.private_ip} >> private_ips.txt"
  }
}
```

Remote-exec provisioner

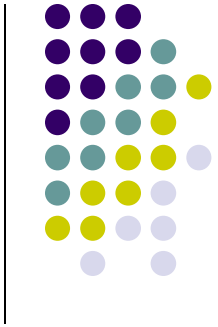


Remote-exec provisioners allow to invoke scripts directly on the remote server.

Let's take an example:

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

  provisioner "remote-exec" {

                    ..............................

  }
}
```

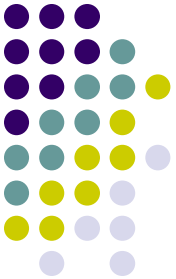
Provisioners



There are two primary types of provisioners:

Types of Provisioners	Description
Creation-Time Provisioner	Creation-time provisioners are only run during creation, not during updating or any other lifecycle If a creation-time provisioner fails, the resource is marked as tainted.
Destroy-Time Provisioner	Destroy provisioners are run before the resource is destroyed.

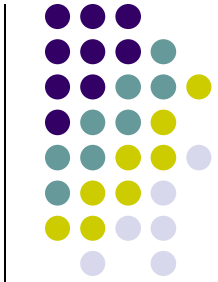
Destroy time provisioner



If when = destroy is specified, the provisioner will run when the resource it is defined within is destroyed.

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

  provisioner "local-exec" {
    when    = destroy
    command = "echo 'Destroy-time provisioner'"
  }
}
```

Local exec provisioner



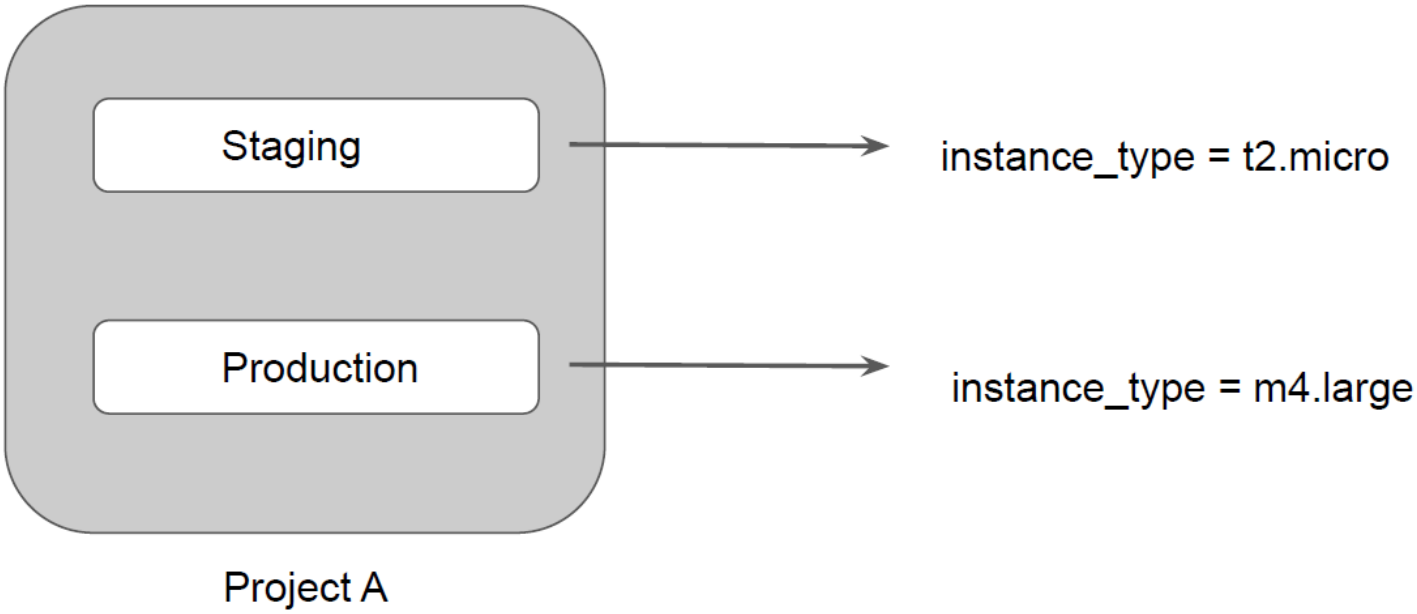
local-exec provisioners allows us to invoke a local executable after the resource is created.

One of the most used approach of local-exec is to run ansible-playbooks on the created server after the resource is created.

```
provisioner "local-exec" {
    command = "echo ${aws_instance.web.private_ip} >> private_ips.txt"
  }
```

Understanding workspace

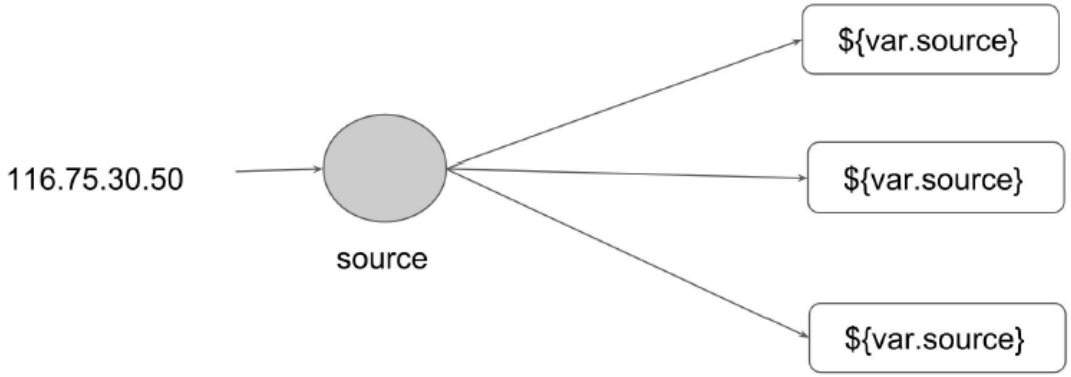
Terraform allows us to have multiple workspaces, with each of the workspace we can have different set of environment variables associated



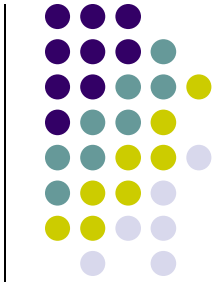
Module and DRY priciple

In software engineering, don't repeat yourself (DRY) is a principle of software development aimed at reducing repetition of software patterns.

In the earlier lecture, we were making static content into variables so that there can be single source of information.



Repeat happens without module



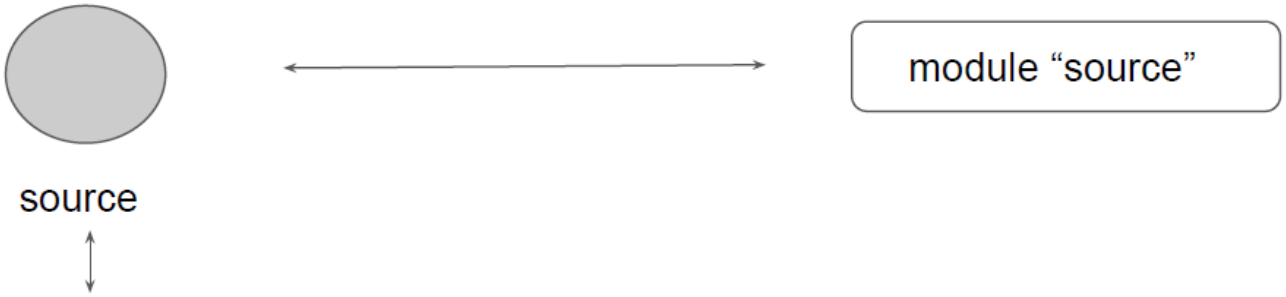
We do repeat multiple times various terraform resources for multiple projects.

Sample EC2 Resource

```
resource "aws_instance" "myweb" {
    ami = "ami-bf5540df"
   instance_type = "t2.micro"
   security_groups = ["default"]
 }
```

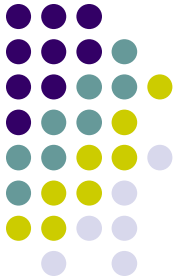
Centralized structure

We can centralize the terraform resources and can call out from TF files whenever required.




```
resource "aws_instance" "myweb" {
    ami = "ami-bf5540df"
   instance_type = "t2.micro"
   security_groups = ["default"]
}
```

Module challenge

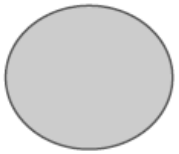


One common need on infrastructure management is to build environments like staging, production with similar setup but keeping environment variables different.

Staging	instance_type = t2.micro
Production	instance_type = m4.large

Module challenge

When we use modules directly, the resources will be replica of code in the module.



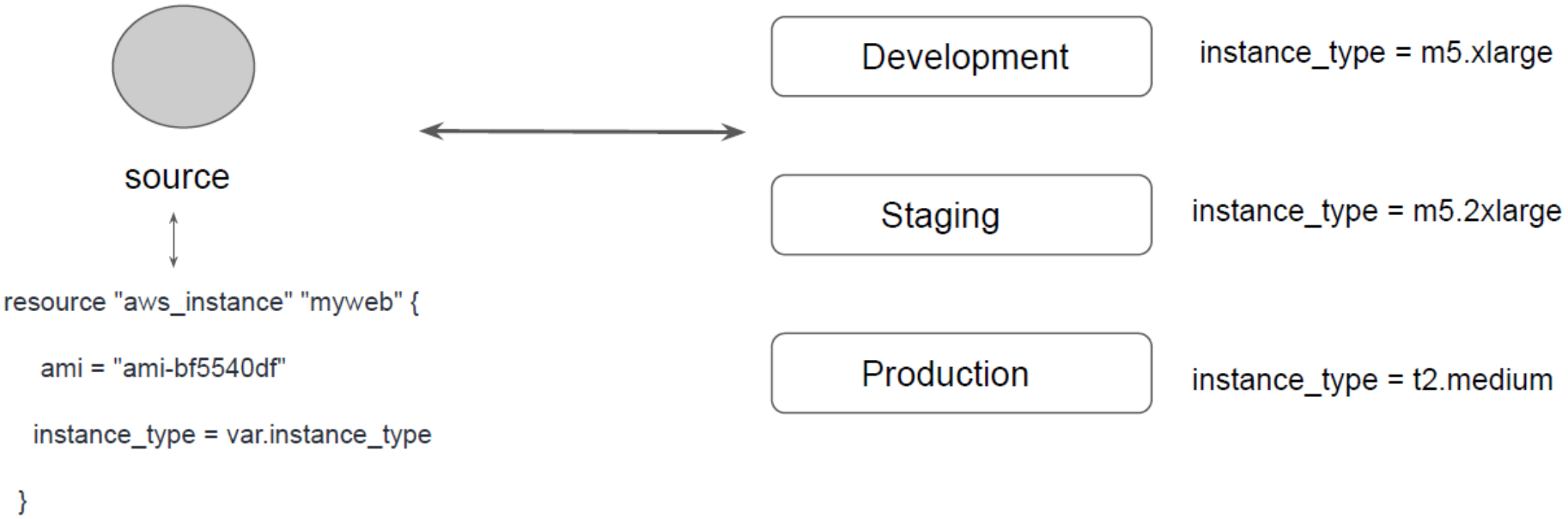
source


```
resource "aws_instance" "myweb" {
    ami = "ami-bf5540df"
   instance_type = "t2.micro"
   security_groups = ["default"]
}
```


Development	t2.micro
Staging	t2.small
Production	m4.large

Using local and challenge

Using variables in Modules can also allow users to override the values which you might not want.



Variable and override by user

There can be many repetitive values in modules and this can make your code difficult to maintain.

You can centralize these using variables but users will be able to override it.

```
resource "aws_security_group" "elb-sg" {
  name        = "myelb-sg"


  ingress {
    description      = "Allow Inbound from Secret Application"
    from_port        = 8443
    to_port          = 8443
    protocol         = "tcp"
    cidr_blocks      = ["0.0.0.0/0"]
  }
```

Hardcoded Port


```
resource "aws_security_group" "elb-sg" {
  name        = "myelb-sg"


  ingress {
    description      = "Allow Inbound from Secret Application"
    from_port        = var.app_port
    to_port          = var.app_port
    protocol         = "tcp"
    cidr_blocks      = ["0.0.0.0/0"]
  }
```

Variable Port

Using locals

Instead of variables, you can make use of locals to assign the values.

You can centralize these using variables but users will be able to override it.

```
resource "aws_security_group" "ec2-sg" {
  name        = "myec2-sg"

  ingress {
    description      = "Allow Inbound from Secret Application"
    from_port        = local.app_port
    to_port          = local.app_port
    protocol         = "tcp"
    cidr_blocks      = ["0.0.0.0/0"]
  }

locals {
  app_port = 8443
}
```

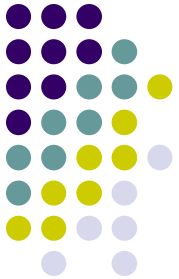
Accessing Child module output

In a parent module, outputs of child modules are available in expressions as module.<MODULE NAME>.<OUTPUT NAME>

```
resource "aws_security_group" "ec2-sg" {
  name        = "myec2-sg"

  ingress {
    description      = "Allow Inbound from Secret Application"
    from_port        = 8433
    to_port          = 8433
    protocol         = "tcp"
    cidr_blocks      = ["0.0.0.0/0"]
  }

}

output "sg_id" {
  value = aws_security_group.ec2-sg.arn
}
```

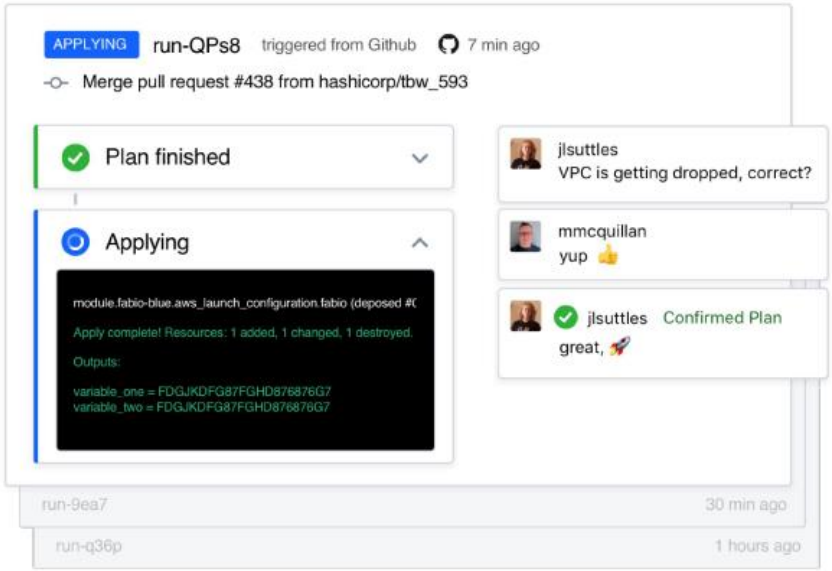
```
module "sgmodule" {
  source = "../../modules/sg"
}

resource "aws_instance" "web" {
  ami           = "ami-0ca285d4c2cda3300"
  instance_type = "t3.micro"
  vpc_security_group_ids = [module.sgmodule.sg_id]
}
```

Day 5



Terraform Cloud

Terraform Cloud manages Terraform runs in a consistent and reliable environment with various features like access controls, private registry for sharing modules, policy controls and others.



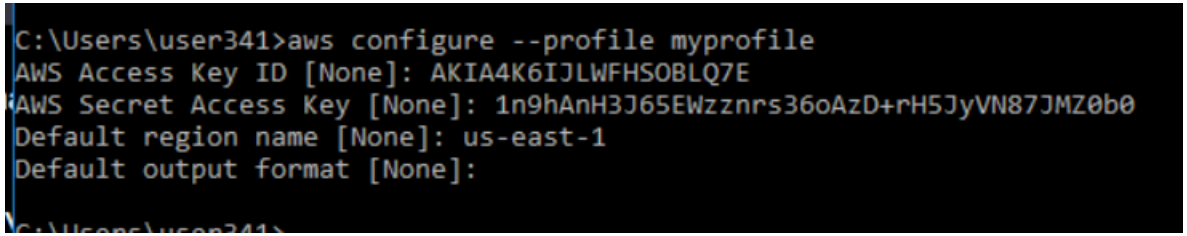
Dynamic block

Terraform are useful for creating multiple instances of a block dynamically based on a list or map of values.

- The `variable "ingress_rules"` defines a list of ingress rules as input, where each rule has `from_port`, `to_port`, `protocol`, and `cidr_blocks` attributes.
- The `aws_security_group` resource uses a dynamic block to dynamically create multiple `ingress` blocks based on the values in `var.ingress_rules`.
- The `for_each` expression iterates over each element in `var.ingress_rules`, and for each element, it creates an `ingress` block with the specified attributes.

This allows you to easily extend or modify the list of ingress rules without having to manually duplicate or modify blocks in your Terraform configuration. Adjust the rules in the `var.ingress_rules` variable according to your needs.

AWS – Profile based credentials



Instead of specifying the profile in the Terraform configuration, you can use the `AWS_PROFILE` environment variable. Open a terminal and run:

bash Copy code

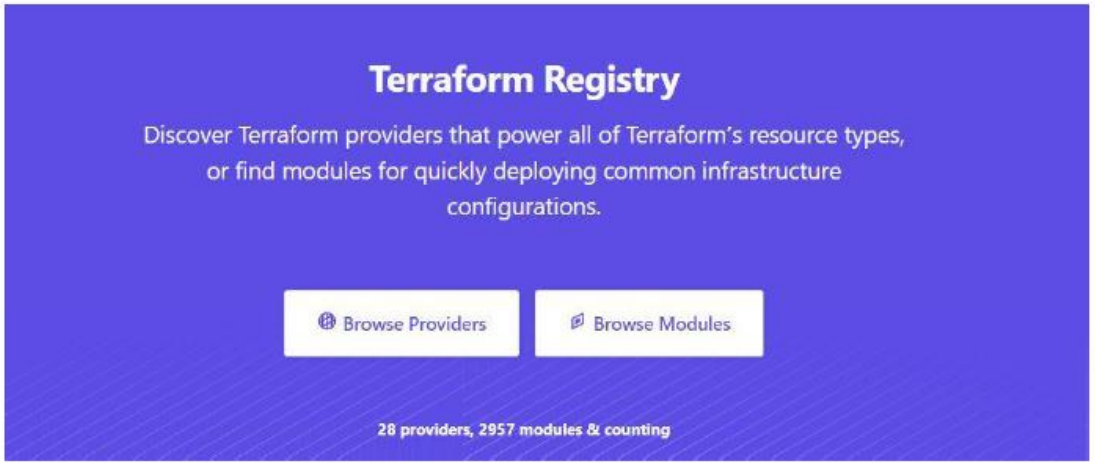
```
export AWS_PROFILE=myprofile
terraform init
terraform apply
```

This way, you don't need to include the `profile` attribute in every provider block.

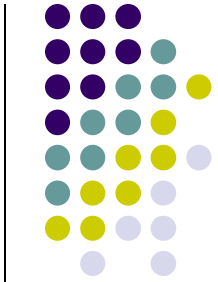
Terraform Registry

The Terraform Registry is a repository of modules written by the Terraform community.

The registry can help you get started with Terraform more quickly



Module location

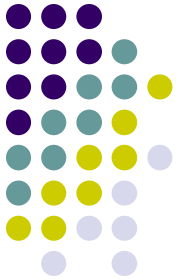


If we intend to use a module, we need to define the path where the module files are present.

The module files can be stored in multiple locations, some of these include:

- Local Path
- GitHub
- Terraform Registry
- S3 Bucket
- HTTP URLs

Verified module in Terraform Registry



Within Terraform Registry, you can find verified modules that are maintained by various third party vendors.

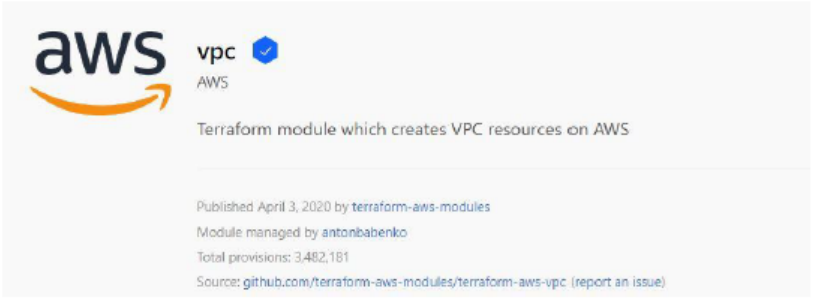
These modules are available for various resources like AWS VPC, RDS, ELB and others.

Verified module Terraform Registry

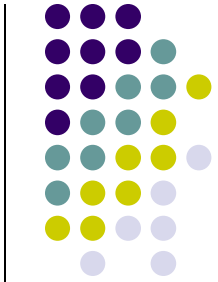
Verified modules are reviewed by HashiCorp and actively maintained by contributors to stay up-to-date and compatible with both Terraform and their respective providers.

The blue verification badge appears next to modules that are verified.

Module verification is currently a manual process restricted to a small group of trusted HashiCorp partners.



Using Registry module



To use Terraform Registry module within the code, we can make use of the source argument that contains the module path.

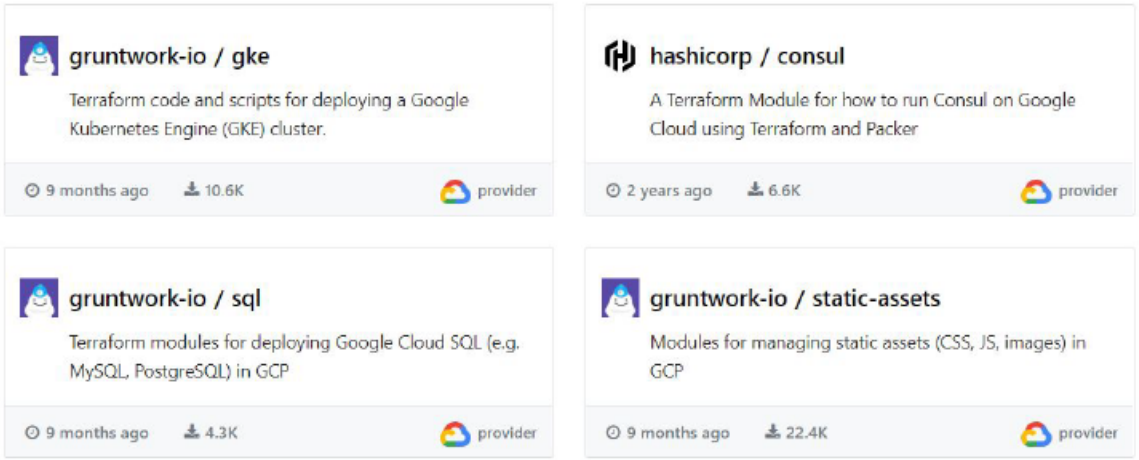
Below code references to the EC2 Instance module within terraform registry.

```
module "ec2-instance" {
  source  = "terraform-aws-modules/ec2-instance/aws"
  version = "2.13.0"
  # insert the 10 required variables here
}
```

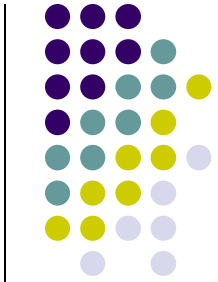
Publishing modules

Anyone can publish and share modules on the Terraform Registry.

Published modules support versioning, automatically generate documentation, allow browsing version histories, show examples and READMEs, and more.



Requirement for publishing Module



Requirement	Description
GitHub	The module must be on GitHub and must be a public repo. This is only a requirement for the public registry.
Named	Module repositories must use this three-part name format terraform-<PROVIDER>-<NAME>
Repository description	The GitHub repository description is used to populate the short description of the module.
Standard module structure	The module must adhere to the standard module structure.
x.y.z tags for releases	The registry uses tags to identify module versions. Release tag names must be a semantic version, which can optionally be prefixed with a v. For example, v1.0.4 and 0.9.2

Standard module structure

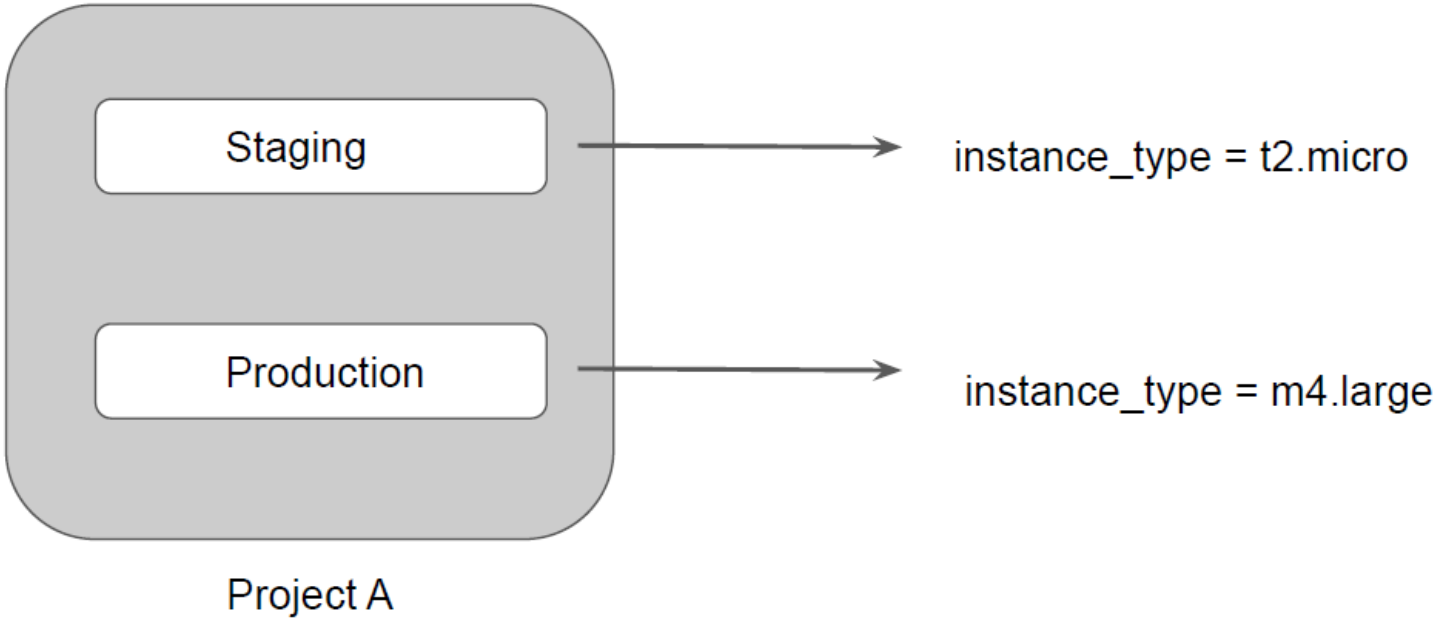
The standard module structure is a file and directory layout that is recommend for reusable modules distributed in separate repositories

```
$ 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/
│   └── .../
```

Workspace

Terraform allows us to have multiple workspaces, with each of the workspace we can have different set of environment variables associated



Terraform Backend

Backends primarily determine where Terraform stores its state.

By default, Terraform implicitly uses a backend called local to store state as a local file on disk.

```
provider "vault" {
  address = "http://127.0.0.1:8200"
}

data "vault_generic_secret" "demo" {
  path = "secret/db_creds"
}

output "vault_secrets" {
  value = data.vault_generic_secret.demo.data_json
  sensitive = "true"
}
```

demo.tf

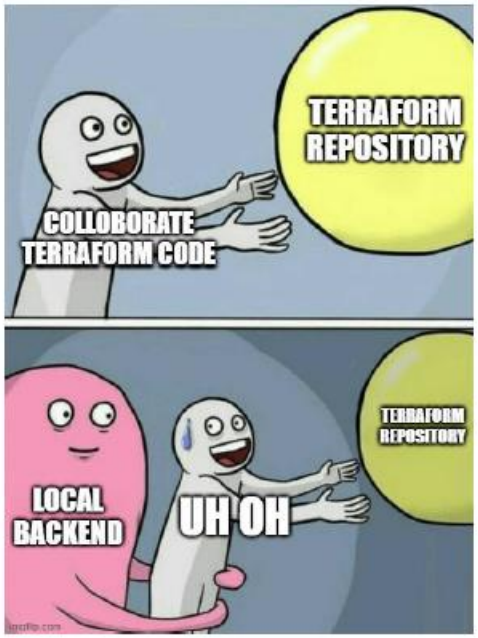
terraform.tfstate

```
{
  "version": 4,
  "terraform_version": "1.1.9",
  "serial": 1,
  "lineage": "f7ba581a-ab47-b03e-2e54-e683a2dc4ba2",
  "outputs": {
    "vault_secrets": {
      "value": "{\"admin\":\"password123\"}",
      "type": "string",
      "sensitive": true
    }
  },
  "resources": [
    {
      "mode": "data",
      "type": "vault_generic_secret",
      "name": "demo",
      "provider": "provider[\"registry.terraform.io/hashicorp/vault\"]",
      "instances": [
```

Challenge with local backend

Nowadays Terraform project is handled and collaborated by an entire team.

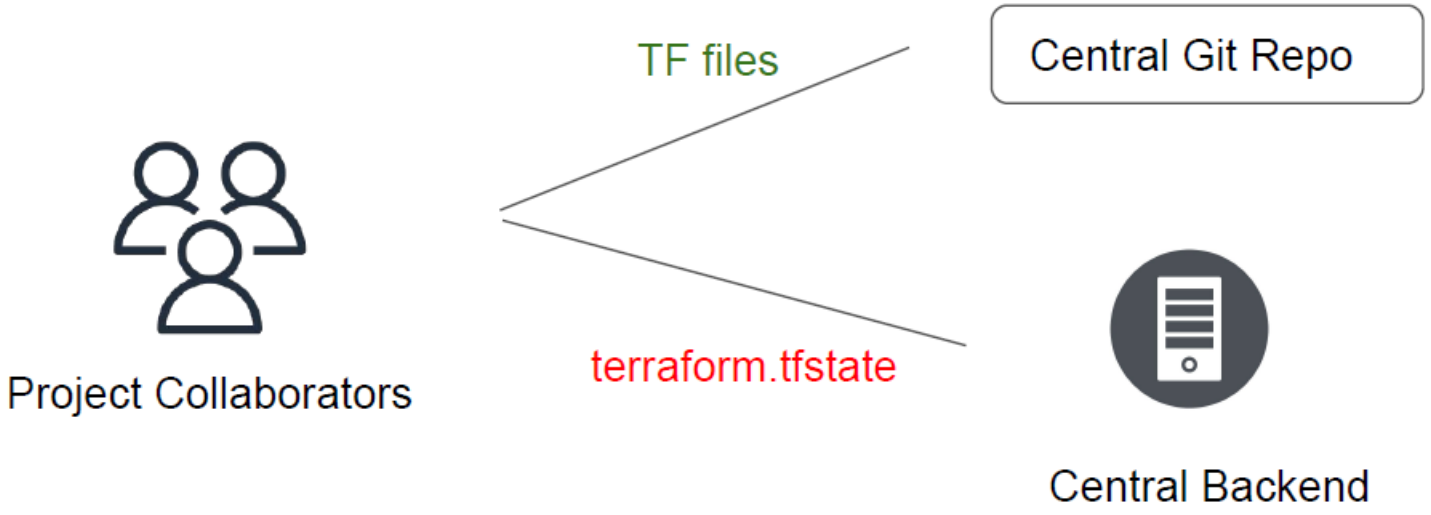
Storing the state file in the local laptop will not allow collaboration.



Desired architecture

Following describes one of the recommended architectures:

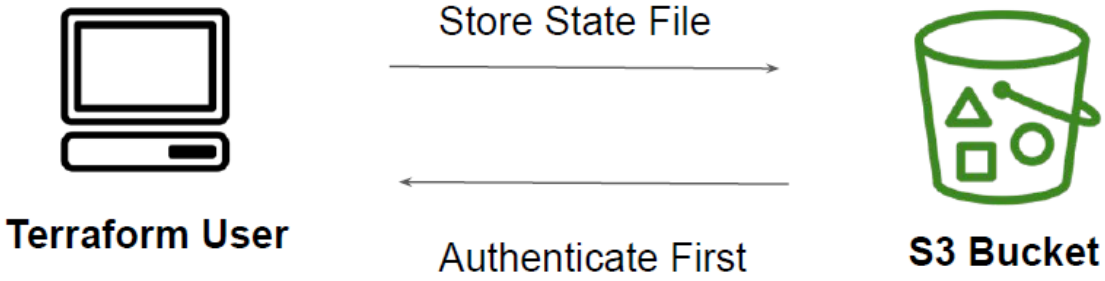
1. The Terraform Code is stored in Git Repository.
2. The State file is stored in a Central backend.



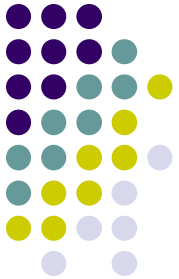
Securing backend

Accessing state in a remote service generally requires some kind of access credentials

Some backends act like plain "remote disks" for state files; others support locking the state while operations are being performed, which helps prevent conflicts and inconsistencies.



Terraform Certification details



Exam

Overview of the basic exam related information.

Assessment Type	Description
Type of Exams	Multiple Choice
Format	Online Proctored
Duration	1 hour
Questions	57
Price	70.50 USD + Taxes
Language	English
Expiration	2 years

Question type

This includes various sub-formats, including:

- True or False
- Multiple Choice
- Fill in the blank

Type of question

Example 1:

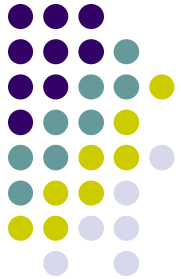
Demo Software stores information in which type of backend?

Online proctored

Important Rules to be followed:

- You are alone in the room
- Your desk and work area are clear
- You are connected to a power source
- No phones or headphones
- No dual monitors
- No leaving your seat
- No talking
- Webcam, speakers, and microphone must remain on throughout the test.
- The proctor must be able to see you for the duration of the test.

Exam



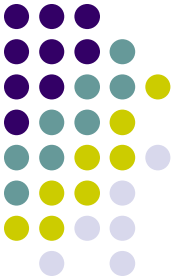
A provider is responsible for understanding API interactions and exposing resources.

Most of the available providers correspond to one cloud or on-premises infrastructure platform, and offer resource types that correspond to each of the features of that platform.

You can explicitly set a specific version of the provider within the provider block.

To upgrade to the latest acceptable version of each provider, run terraform init -upgrade

Exam

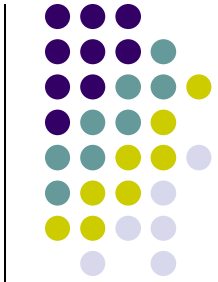


You can have multiple provider instance with the help of alias

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

provider "aws" {
  alias  = "west"
  region = "us-west-2"
  }
```

The provider block without alias set is known as the default provider configuration. When an alias is set, it creates an additional provider configuration.

Exam



The terraform init command is used to initialize a working directory containing Terraform configuration files.

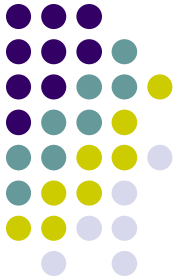
During init, the configuration is searched for module blocks, and the source code for referenced modules is retrieved from the locations given in their source arguments.

Terraform must initialize the provider before it can be used.

Initialization downloads and installs the provider's plugin so that it can later be executed.

It will not create any sample files like example.tf

Exam



The terraform plan command is used to create an execution plan.

It will not modify things in infrastructure.

Terraform performs a refresh, unless explicitly disabled, and then determines what actions are necessary to achieve the desired state specified in the configuration files.

This command is a convenient way to check whether the execution plan for a set of changes matches your expectations without making any changes to real resources or to the state.

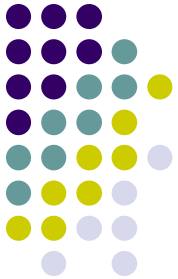
Exam

The terraform apply command is used to apply the changes required to reach the desired state of the configuration.

Terraform apply will also write data to the terraform.tfstate file.

Once apply is completed, resources are immediately available.

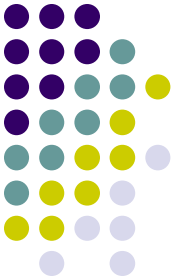
Exam



The terraform refresh command is used to reconcile the state Terraform knows about (via its state file) with the real-world infrastructure.

This does not modify infrastructure but does modify the state file.

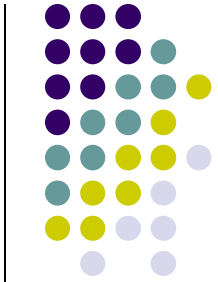
Exam



The terraform destroy command is used to destroy the Terraform-managed infrastructure.

terraform destroy command is not the only command through which infrastructure can be destroyed.

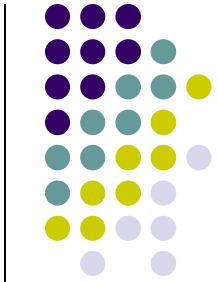
Exam



The terraform fmt command is used to rewrite Terraform configuration files to a canonical format and style.

For use-case, where the all configuration written by team members needs to have a proper style of code, terraform fmt can be used.

Exam



The terraform validate command validates the configuration files in a directory.

Validate runs checks that verify whether a configuration is syntactically valid and thus primarily useful for general verification of reusable modules, including the correctness of attribute names and value types.

It is safe to run this command automatically, for example, as a post-save check in a text editor or as a test step for a reusable module in a CI system. It can run before terraform plan.

Validation requires an initialized working directory with any referenced plugins and modules installed

Exam

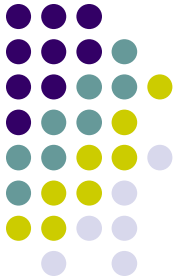
Provisioners can be used to model specific actions on the local machine or on a remote machine in order to prepare servers or other infrastructure objects for service.

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

- Provisioners are inside the resource block.
- Have an overview of local and remote provisioner

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

  provisioner "local-exec" {
    command = "echo The server's IP address is ${self.private_ip}"
  }
}
```

Exam



Terraform has detailed logs that can be enabled by setting the **TF_LOG** environment variable to any value.

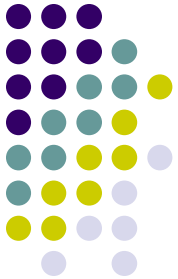
You can set TF_LOG to one of the log levels TRACE, DEBUG, INFO, WARN or ERROR to change the verbosity of the logs.

Example:

TF_LOG=TRACE

To persist logged output, you can set TF_LOG_PATH

Exam



Terraform is able to import existing infrastructure.

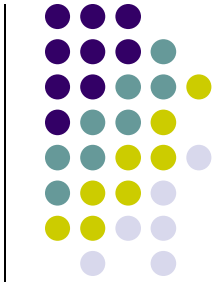
This allows you take resources that you've created by some other means and bring it under Terraform management.

The current implementation of Terraform import can only import resources into the state. It does not generate configuration.

Because of this, prior to running terraform import, it is necessary to write a resource configuration block manually for the resource, to which the imported object will be mapped.

terraform import aws_instance.myec2 instance-id

Exam



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

The expression of a local value can refer to other locals, but as usual reference cycles are not allowed. That is, a local cannot refer to itself or to a variable that refers (directly or indirectly) back to it.

It's recommended to group together logically-related local values into a single block, particularly if they depend on each other.

Exam

Terraform allows us to have multiple workspaces; with each of the workspaces, we can have a different set of environment variables associated.

Workspaces allow multiple state files of a single configuration.