




# Terraform Foundation



## Overview of the Course

This is a certification specific course with primary intention to cover all the topics of HashiCorp Certified Terraform Associate certification.

The arrangement of topics within the course is a little different from the exam blueprint to ensure the course remains beginner-friendly.



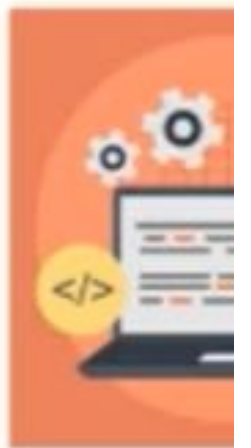
# IAC Tools

DevOps = Developers

# Exploring Toolsets

There are various types of tools that can allow you to deploy infrastructure as code

- Terraform
- CloudFormation
- Heat
- Ansible
- SaltStack
- Chef, Puppet and others






# Configuration Management vs Infrastructure Orchestration

↳

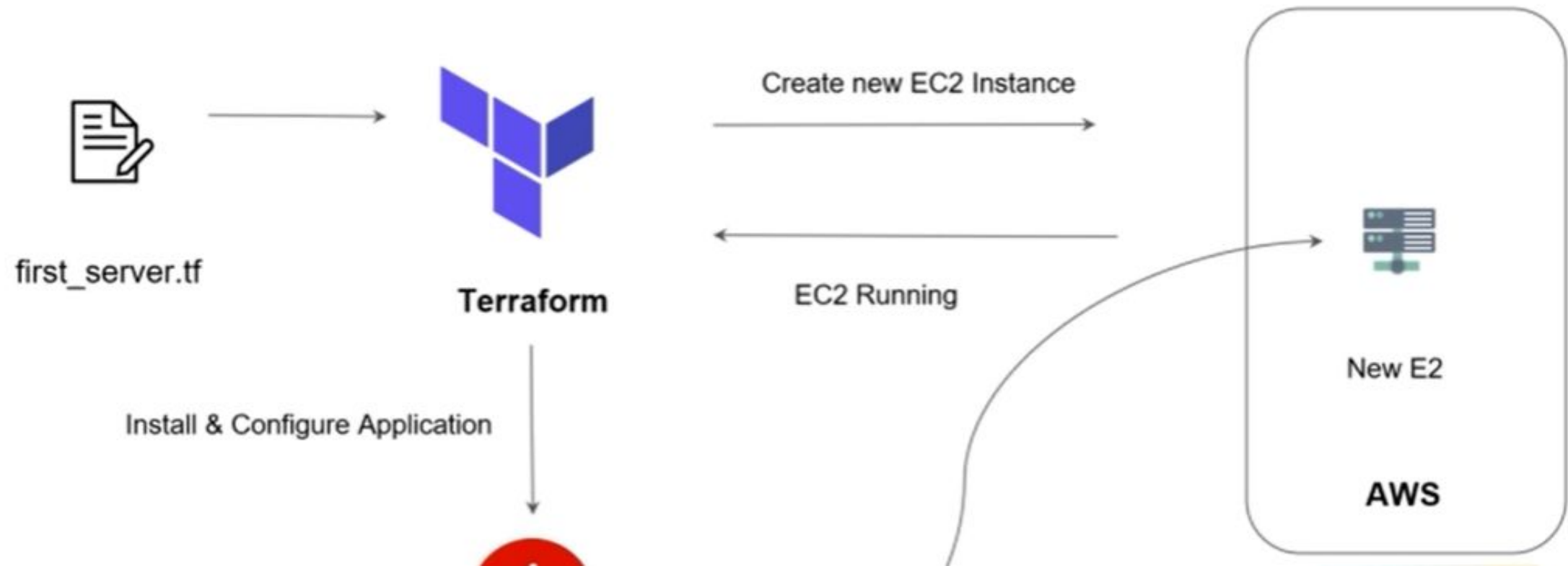
Ansible, Chef, Puppet are configuration management tools which means that they are primarily designed to install and manage software on existing servers.

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



# Which tool to choose ?

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 !!!



# Supported Platforms

Terraform works on multiple platforms, these includes:

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

## Terraform Installation - Mac & Linux

There are two primary steps required to install terraform in Mac and Linux

- 1) Download the Terraform Binary File.
- 2) Move it in the right path.



# Setting up the Lab

Let's start Rolling !



# Let's Start

i) Register an AWS Account.



ii) Begin the course





# Providers & Resources

Terraform in detail

---





# Providers

Terraform supports multiple providers.

We have to specify the provider details for which we want to launch the infrastructure for.

With provider, we also have to add the tokens which will be used for authentication.

On adding a provider, `terraform init` will download plugins associated with the provider.



# Resources

Resources are the reference to the individual services which the provider has to offer

Example:

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



# Terraform State File

Terraform in detail







## State File

Terraform stores the state of the infrastructure that is being created from the TF files.

This state allows terraform to map real world resource to your existing configuration.





EC2 Instance



```
instance-id = i-23a23sdsd
Instance-type = t2.micro
sg = sg-3dse24ws
private ip = 172.31.28.132
volume-type = gp2
volume_size = 8 GB
public-ip = 34.215.116.193
```

terraform.tfstate — C:\Users\Zeal Vora\Desktop\kplabs-terraform — Atom

File Edit View Selection Find Packages Help

Project

- ▼ kplabs-terraform
  - > .terraform
    - do\_droplet.tf
    - first\_ec2.tf
    - terraform.tfstate
    - terraform.tfstate.backup

```
1 {
2   "version": 4,
3   "terraform_version": "0.12.5",
4   "serial": 9,
5   "lineage": "14cdf22e-8aad-47ba-936b-153f65b72894",
6   "outputs": {},
7   "resources": []
8 }
```



# Provider Versioning

Terraform in detail



Provider plugins are released separately from Terraform itself.

They have different set of version numbers.

.



Version 1



Version 2

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.

```
provider "aws" {  
  region    = "us-west-2"  
  version   = "2.7"  
}
```

# Arguments for Specifying provider

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

Version Number Arguments	Description
$\geq 1.0$	Greater than equal to the version
$\leq 1.0$	Less than equal to the version
$\sim > 2.0$	Any version in the 2.X range.
$\geq 2.10, \leq 2.30$	Any version between 2.10 and 2.30



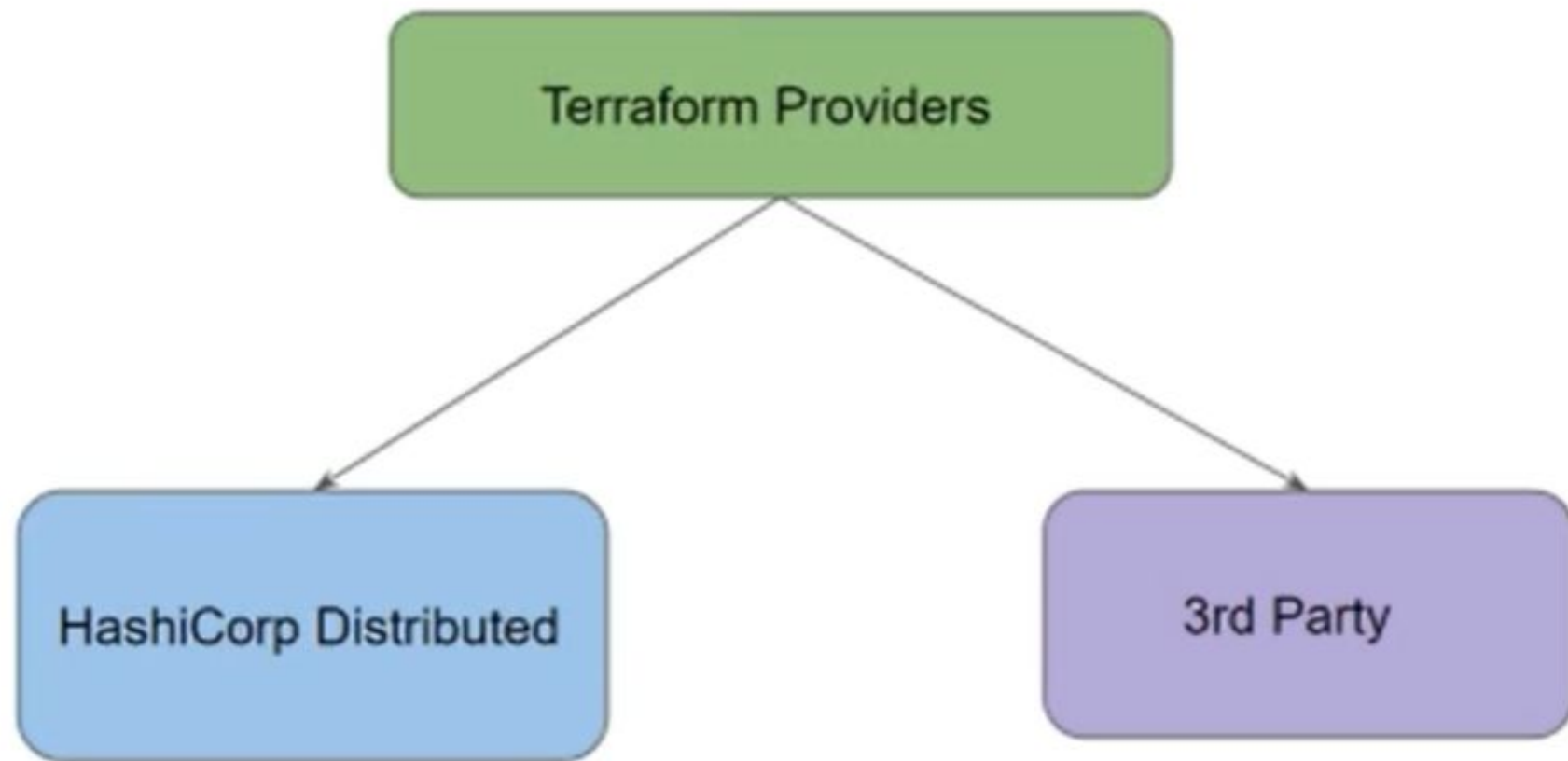
# Types of Terraform Providers

Terraform in detail





There are two major categories for terraform providers.





## Overview of Providers

HashiCorp Distributed providers can be downloaded automatically during terraform init.

terraform init cannot automatically download providers that are not distributed by HashiCorp





## Overview of 3rd Party Providers

It can happen that the official provider do not support a specific functionality.

Some organizations might have their proprietary platform for which they want to use Terraform.

For such cases, individuals can decide to develop / use 3rd party providers.



Operating system	User plugins directory
Windows	%APPDATA%\terraform.d\plugins
All other systems	~/.terraform.d/plugins

## Configuring 3rd Party Provider

Third-party providers must be manually installed, since terraform init cannot automatically download them.

Install third-party providers by placing their plugin executables in the user plugins directory.



# Lecture Format - Terraform Course

Terraform in detail





# Attributes & Output Values



# Understanding Attributes

Terraform has capability to output the attribute of a resource with the output values.

Example:

```
ec2_public_ip    = 35.161.21.197
```

```
bucket_identifier = terraform-test-kplabs.s3.amazonaws.com
```



## Attributes are important

An outputted attribute can not only be used for the user reference but it can also act as an input to other resources being created via terraform

Let's understand this with an example:

After EIP gets created, its IP address should automatically get whitelisted in the security group.



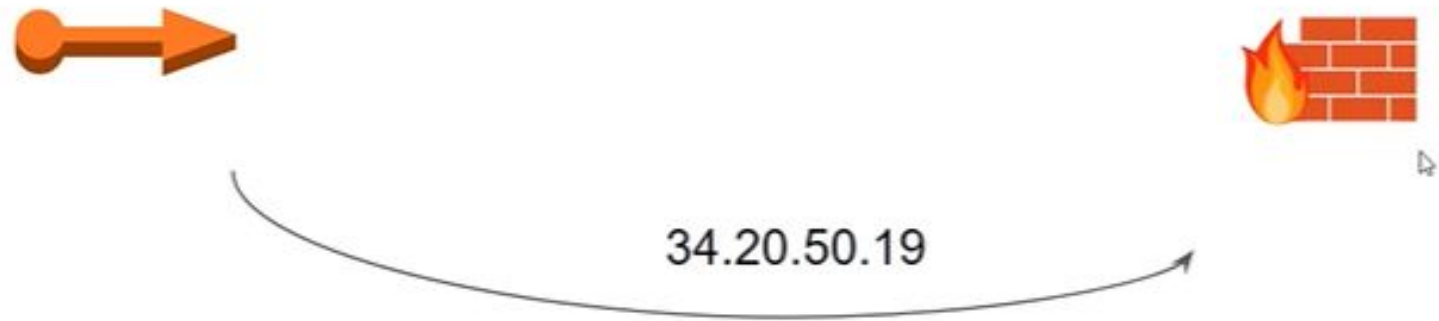


# Referencing Cross-Resource Attributes

## Example 1: EIP and EC2 Instance



## Example 2: EIP and Security Group





# Terraform Variables

Terraform in detail



# Static = Work

Repeated static values can create more work in the future.



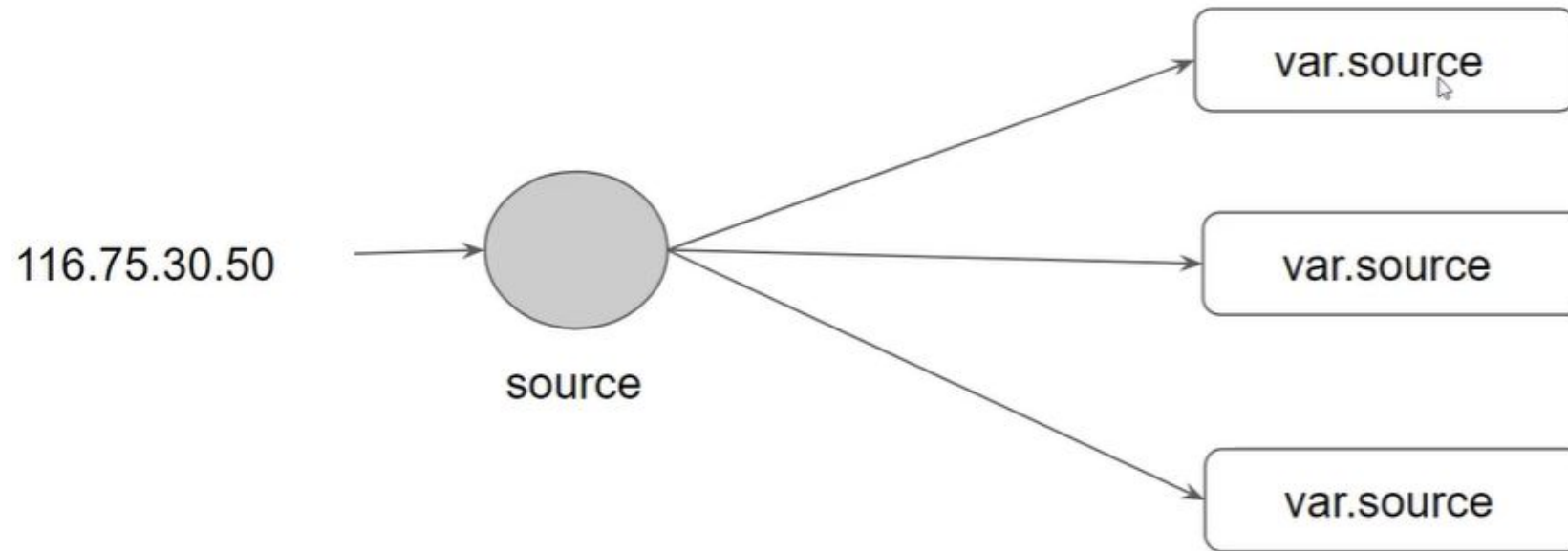
Project A



Project B

# Variables are good

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





# Approaches to Variable 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





# Data Types for Variables





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.



## Example Use-Case

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.

<b>variables.tf</b>	<b>terraform.tfvars</b>
variable "instance_name" {}	instance_name="john-123"

## Example Use-Case

Every employee in Medium Corp is assigned a Identification Number.

Any EC2 instance that employee creates should be created using the identification num

variables.tf	terraform.tfvars
<pre>variable "instance_name" {   type=number }</pre>	<pre>instance_name="john-123"</pre>

# Overview of Data Types

Type Keywords	Description
<code>string</code>	Sequence of Unicode characters representing some text,
<code>list</code>	Sequential list of values identified by their position. Sta [ <code>"mumbai"</code> , <code>"singapore"</code> , <code>"usa"</code> ]
<code>map</code>	a group of values identified by named labels, lil { <code>name = "Mabel"</code> , <code>age = 52</code> }.
<code>number</code>	Example: 200



# Count Parameter

Terraform in detail



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"  
}
```

## Overview of 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.

# Understanding 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/"  
}
```

## Understanding Challenge with Count

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/"  
}
```




## Understanding Challenge with Default Count Index

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.





# Conditional Expression

Terraform in detail



# Overview of Conditional Expression

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

Syntax of Conditional expression:

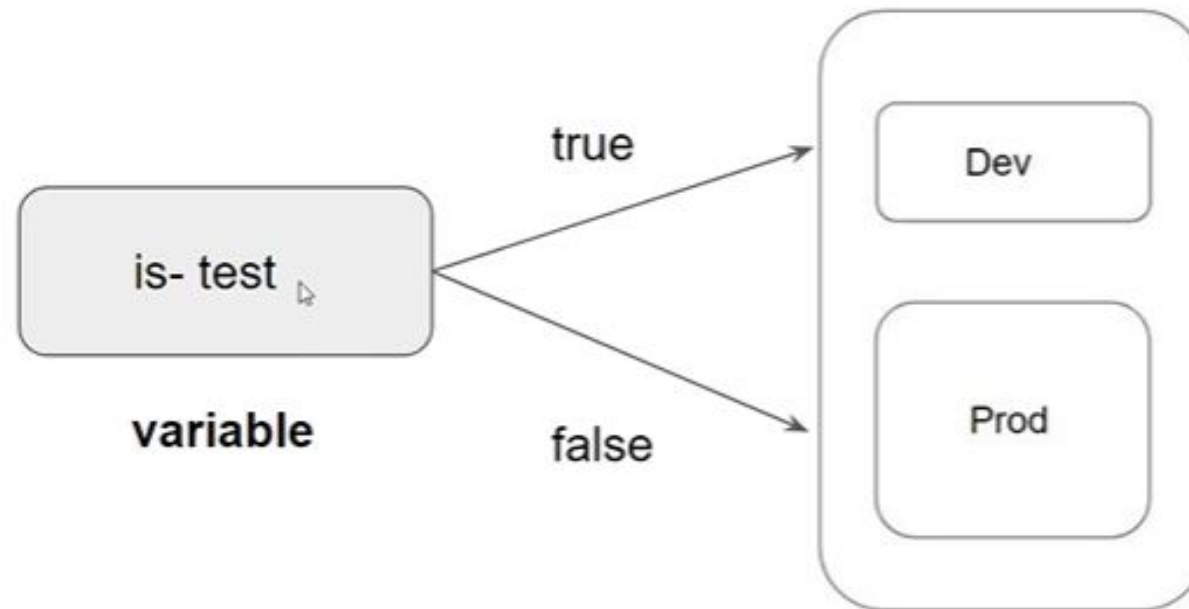
```
condition ? true_val : false_val
```

If condition is true then the result is true\_val. If condition is false then the result is false\_val.

# Example of Conditional Expression

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

Terraform in detail








## Overview of 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 Support for Expression

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

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




## Important Pointers for Local Values

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 Functions

Terraform in detail



# Overview of Terraform Functions

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)`

## 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



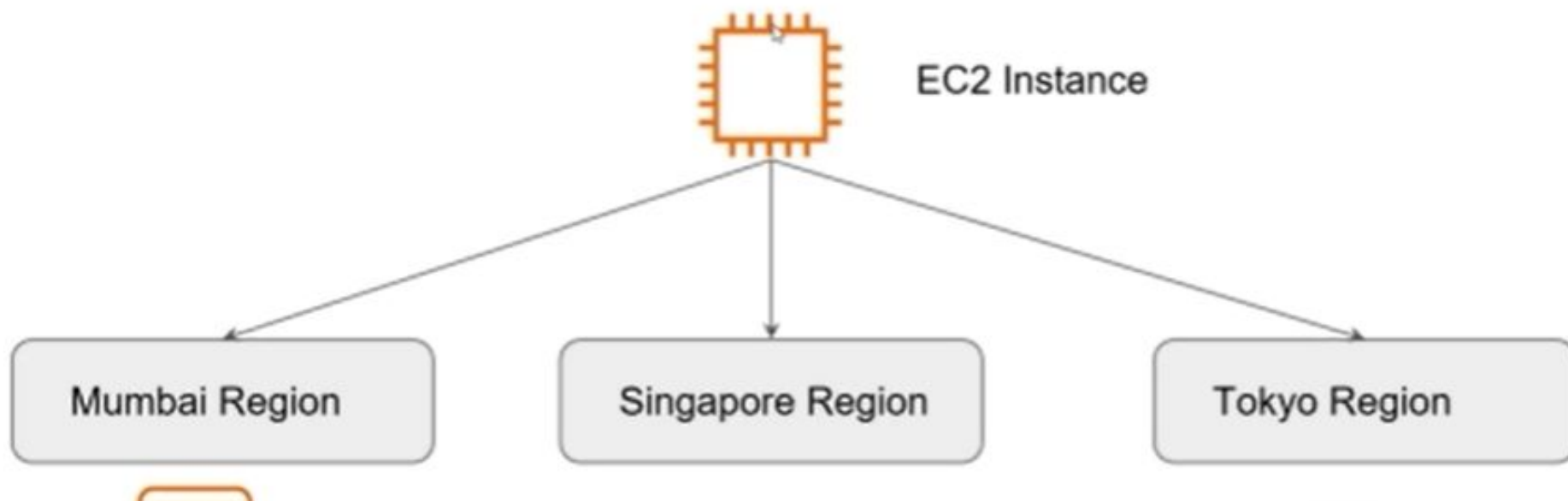
# Data Sources

Terraform in detail



## Overview of Data Sources

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





## Data Source Code

- 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 in detail



# Overview of 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
```



## Important Pointers

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.





# Terraform Format

Terraform in detail



## Importance of 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"  
}
```

**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 in detail





# Overview of 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.



# Load Order & Semantics

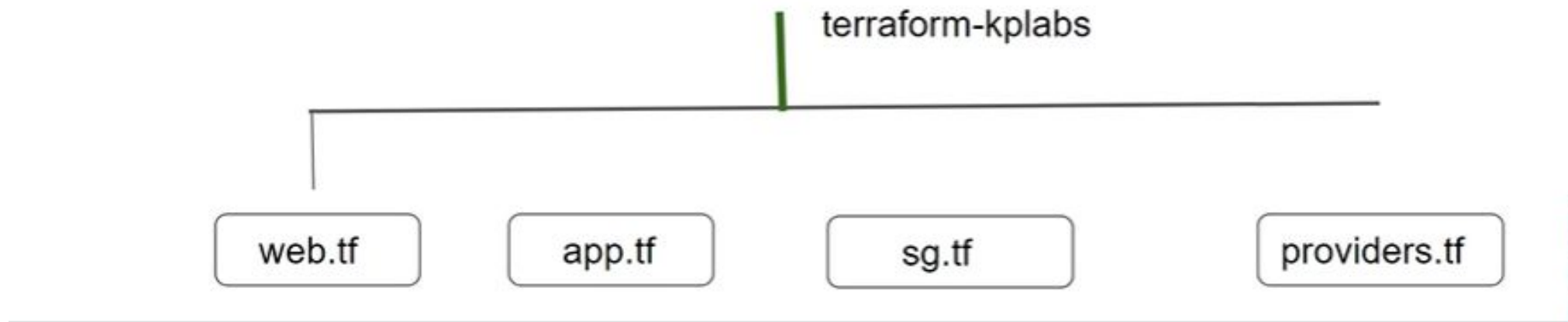
Terraform in detail



# Understanding Semantics

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

Terraform In Depth



# Understanding the Challenge

In many of the use-cases, there are repeatable nested blocks that needs to be defined.

This can lead to a long code and it can be difficult to manage in a longer time.

```
ingress {  
  from_port    = 9200  
  to_port      = 9200  
  protocol     = "tcp"  
  cidr_blocks  = ["0.0.0.0/0"]  
}
```

```
ingress {  
  from_port    = 8300  
  to_port      = 8300  
  protocol     = "tcp"  
  cidr_blocks  = ["0.0.0.0/0"]  
}
```

## Dynamic Blocks

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"]  
  }  
}
```

```
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"]
  }
}
```

## Iterators

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).



# Tainting Resources

Terraform In Detail







Lots of manual changes



New Resource

## Terraform taint

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 The Changes to Terraform / Delete & Recreate the resource

## Overview of Terraform Taint

The terraform taint command manually marks a Terraform-managed resource as tainted, forcing it to be destroyed and recreated on the next apply.



**Terraform**

Taint Resource



**New Resource**

## Important Pointers

This command will not modify infrastructure, but does modify the state file in order to mark a resource as tainted.

Once a resource is marked as tainted, the next plan will show that the resource will be destroyed and recreated and the next apply will implement this change.

Note that tainting a resource for recreation may affect resources that depend on the newly tainted resource.



# Splat Expression

## Terraform Expressions



# Overview of 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

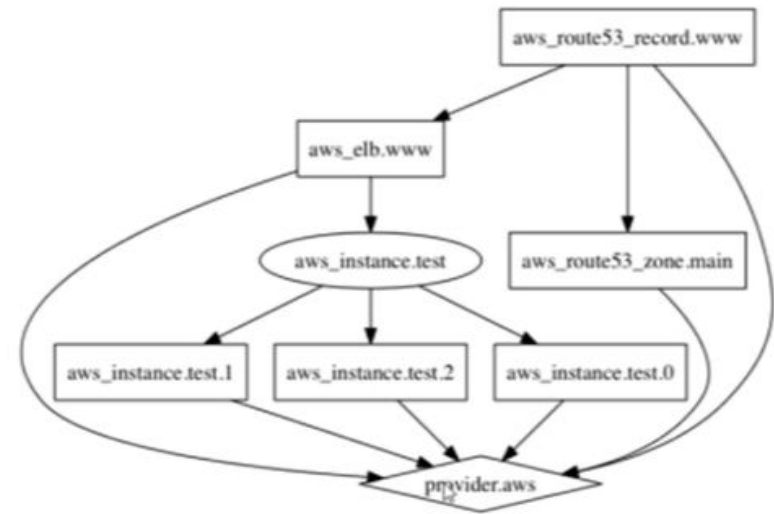
Terraform In Detail



## Overview of 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.





# Saving Terraform Plan to a File

Terraform In Detail





## Terraform Plan 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
```



# Terraform Output

Terraform in detail



# Terraform Output

The terraform output command is used to extract the value of an output variable from the state file.

```
C:\Users\Zeal Vora\Desktop\terraform\terraform output>terraform output iam_names  
[  
  "iamuser.0",  
  "iamuser.1",  
  "iamuser.2",  
]
```



# Terraform Settings

Terraform in detail



## Overview of Terraform Settings

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

Terraform settings are gathered together into `terraform` blocks:

```
terraform {  
  # ...  
}
```

## Setting 1 - Terraform Version

The `required_version` setting accepts a version constraint string, which specifies which versions of Terraform can be used with your configuration.

If the running version of Terraform doesn't match the constraints specified, Terraform will produce an error and exit without taking any further actions.

```
terraform {  
  required_version = "> 0.12.0"  
}
```

## Setting 2 - Provider Version

The `required_providers` block specifies all of the providers required by the current module, mapping each local provider name to a source address and a version constraint.

```
terraform {  
  required_providers {  
    mycloud = {  
      source  = "mycorp/mycloud"  
      version = "~> 1.0"  
    }  
  }  
}
```



# Dealing with Larger Infrastructure

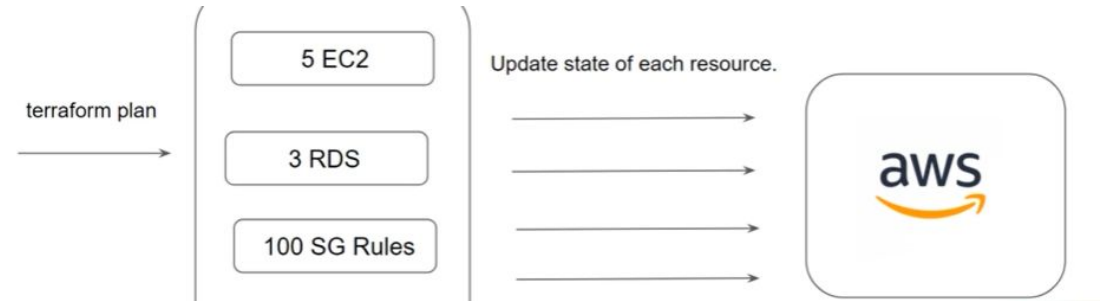
Terraform in detail





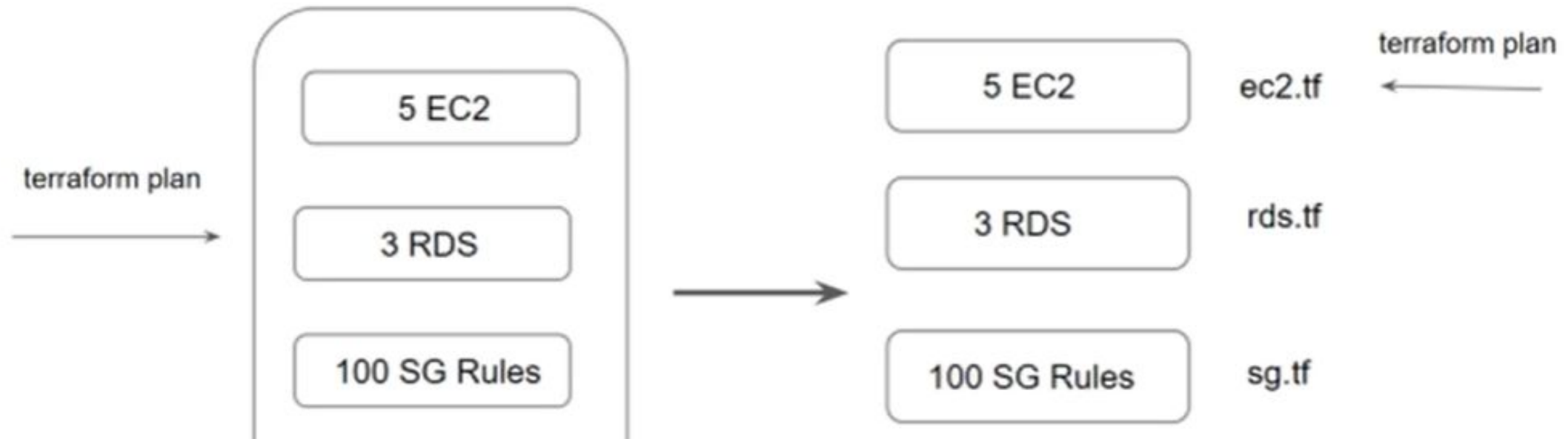
## Challenges with Larger Infrastructure

When you have a larger infrastructure, you will face issue related to API limits for a provider.



## Dealing With Larger Infrastructure

Switch to smaller configuration where each can be applied independently.



## Slow Down, My Man

We can prevent terraform from querying the current state during operations like terraform plan.

This can be achieved with the `-refresh=false` flag



## Specify the Target

The `-target=resource` flag can be used to target a specific resource.

Generally used as a means to operate on isolated portions of very large configurations

`terraform plan -target=ec2`



5 EC2

3 RDS

100 SG Rules



# DRY Principle

Software Engineering

## Understanding DRY Approach

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.

nciple

## We are repeating resource code

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.

