

Terraform is an infrastructure as code tool that lets you build, change, and version cloud and on-prem resources safely and efficiently.

Terraform can provide support with multi-cloud via having a single workflow for every cloud.

It's very similar to tools such as CloudFormation, which you would use to automate your AWS infrastructure, but you can only use that on AWS. With Terraform, you can use it on other cloud platforms as well

Terraform Core concepts

Below are the core concepts/terminologies used in Terraform:

- **Variables:** Also used as input-variables, it is key-value pair used by Terraform modules to allow customization.
- **Provider:** It is a plugin to interact with APIs of service and access its related resources.
- **Module:** It is a folder with Terraform templates where all the configurations are defined
- **State:** It consists of cached information about the infrastructure managed by Terraform and the related configurations.
- **Resources:** It refers to a block of one or more infrastructure objects (compute instances, virtual networks, etc.), which are used in configuring and managing the infrastructure.
- **Data Source:** It is implemented by providers to return information on external objects to terraform.
- **Output Values:** These are return values of a terraform module that can be used by other configurations.
- **Plan:** It is one of the stages where it determines what needs to be created, updated, or destroyed to move from real/current state of the infrastructure to the desired state.
- **Apply:** It is one of the stages where it applies the changes real/current state of the infrastructure in order to move to the desired state.

Terraform Lifecycle

Terraform lifecycle consists of – **init**, **plan**, **apply**, and **destroy**.



- Terraform init initializes the working directory which consists of all the configuration files
- Terraform plan is used to create an execution plan to reach a desired state of the infrastructure. Changes in the configuration files are done in order to achieve the desired state.
- Terraform apply then makes the changes in the infrastructure as defined in the plan, and the infrastructure comes to the desired state. {Deploy the infrastructure as per the code}
- Terraform destroy is used to delete all the old infrastructure resources, which are marked tainted after the apply phase.

Installation on Linux:

[Install Terraform | Terraform | HashiCorp Developer](#)

→sudo yum install -y yum-utils

→sudo yum-config-manager --add-repo
<https://rpm.releases.hashicorp.com/AmazonLinux/hashicorp.repo>

→sudo yum -y install terraform

→terraform --version

-----*****-----

Authentication and Authorization:

Authentication is the process of verifying who the user is.

Authorization is the process of verifying what they have access to.

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

Depending upon the provider environment, the type of access credentials would change.

Ex: In AWS → Access keys and secret keys, In github → tokens, In kubernetes → Kubeconfigfile

This can be done in two ways.

1st way → create IAM user, get credentials for user

→logout and login to Aws console as IAM user

→now generate AWS access key and secret key credentials

Now configure the Authentication and authorization in the Ec2 instance where terraform is installed.

Command→aws configure

Enter the access ID, secret key, default region and format is JSON or table

```
[root@ip-172-31-15-149 ~]# aws configure
AWS Access Key ID [None]: AKIA4QUKASHDU4DO3HGI
AWS Secret Access Key [None]: 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
Default region name [None]: us-west-2
Default output format [None]: table
```

*****-----*****

2nd way is → create an IAM role

In Trusted entity type select AWS service and Use case select EC2 as we have installed terraform on EC2

The screenshot shows the 'Trusted entity type' selection screen in the AWS IAM console. On the left, a sidebar indicates 'Step 2: Add permissions' and 'Step 3: Name, review, and create'. The main area is titled 'Trusted entity type' and contains five radio button options: 'AWS service' (selected), 'AWS account', 'Web identity', 'SAML 2.0 federation', and 'Custom trust policy'. Each option has a brief description of its function. Below these options is a 'Use case' section with the text 'Allow an AWS service like EC2, Lambda, or others to perform actions in this account.'

In the permissions, choose Administrator access

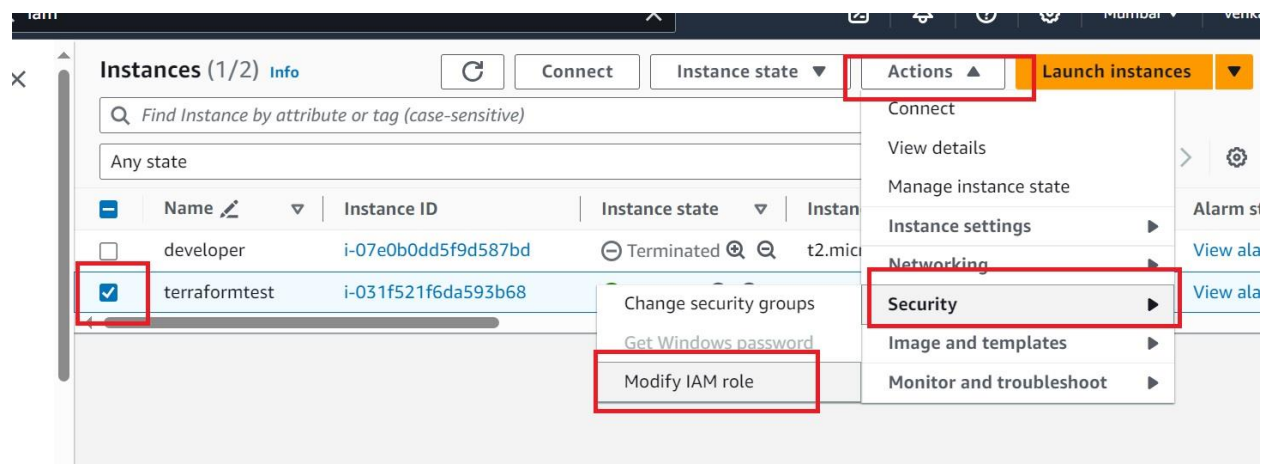
The screenshot shows the 'Permissions policies' selection screen in the AWS IAM console. The title is 'Permissions policies (1/910)' with an 'Info' link. Below the title is a search bar containing 'admin' and a 'Filter by Type' dropdown set to 'All types', showing '37 matches'. A table lists three policies: 'AdministratorAccess' (checked), 'AdministratorAccess-Amplify', and 'AdministratorAccess-AWSElasticBeanstalk'. The table has columns for 'Policy name', 'Type', and 'Description'.

Policy name	Type	Description
<input checked="" type="checkbox"/> AdministratorAccess	AWS managed - job func...	Provides full a
<input type="checkbox"/> AdministratorAccess-Amplify	AWS managed	Grants account
<input type="checkbox"/> AdministratorAccess-AWSElasticBeanstalk	AWS managed	Grants account

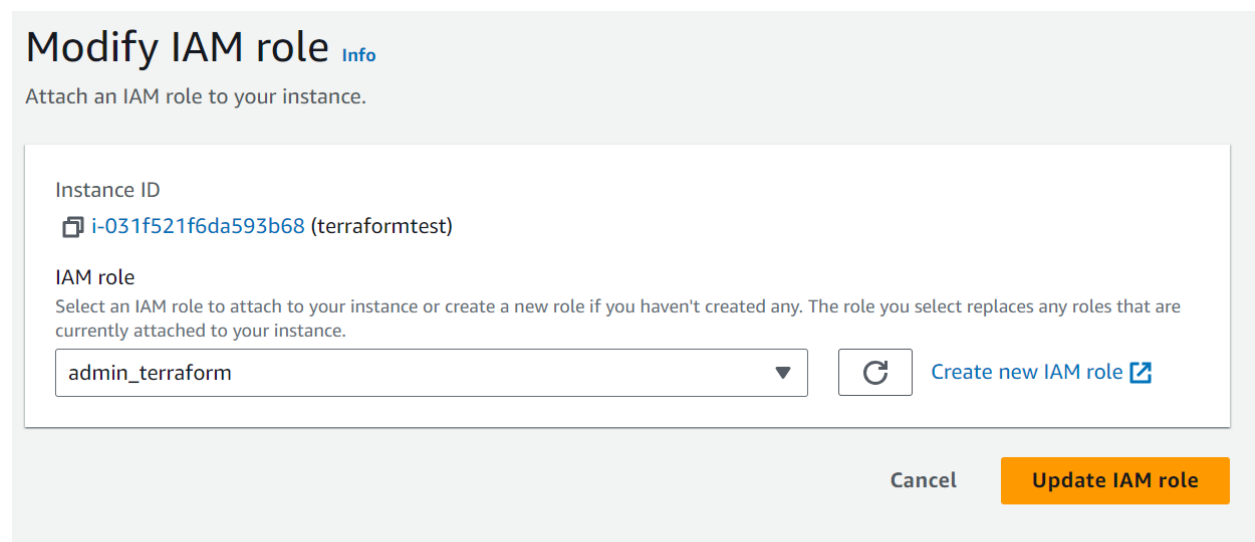
Give the role name and description

The screenshot shows the 'Role details' form in the AWS IAM console. It has two main sections: 'Role name' and 'Description'. The 'Role name' section has a text input field containing 'admin_terraform' and a note: 'Maximum 64 characters. Use alphanumeric and '+', '@', '-' characters.' The 'Description' section has a larger text input field containing 'admin_terraform' and a note: 'Maximum 1000 characters. Use alphanumeric and '+', '@', '-' characters.'

Now go the EC2 instance where terraform is installed. Select the actions→security→Modify IAM role



Now select the IAM role we have created and update it



Now we will be able to create infra using terraform installed in Ec2

-----*****-----

Terraform file mainly consists of →Provider, region →Resource configuration →Variables

Providers: A provider is a plugin that enables interaction with an API. This includes Cloud providers and Software-as-a-service providers.

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

Simply it refers to the cloud platform or cloud provider (AWS or Azure or GCP) we are going to work on.

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

Ex: provider "aws"

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.

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 Contributor

Resource

Resource block describes one or more infrastructure object associated with the provider. Each resource block describes one or more infrastructure objects, such as virtual networks, compute instances, or higher-level components such as DNS records.

Resource block requires two parameters. Resource name and Object/Instance name.

Ex: resource "aws_instance" here aws is resource and instance is object name

resource "aws_alb" here aws is resource and alb is object

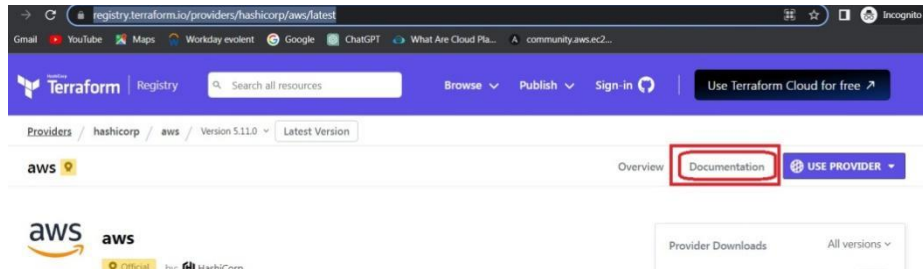
inside resource block we declare its parameters.

These are key-value pairs that define the properties and configuration settings of the resource. The attributes and parameters available depend on the specific resource type being declared.

Creating an EC2 instance using Terraform

Make a directory for terraform. In that directory create a file myfirst.tf (extension is .tf indicates terraform). Now open <https://registry.terraform.io/providers/hashicorp/aws/latest>

Go to documentation

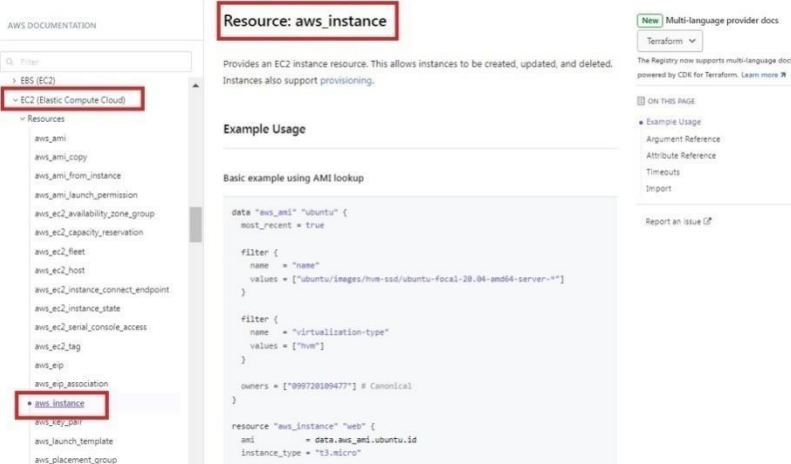


```
provider "aws" {  
  region = "us-west-2"  
}
```

Replace the desired region and

Go to documentation

On the left-hand side available resources, select Ec2 → aws_instance



We have the reference code, and select the necessary parameters

```
resource "aws_instance" "name" {  
  ami = "ami-odcc1e21636832c5d" →ami id of respective region  
  instance_type = "t2.micro" →configuration  
}
```

When we try to create ec2 instance **manually**, at the time of selecting ami we can see the ami id copy that id here.

Now our myfirst.tf contains

```
provider "aws" {  
  region = "us-west-2"  
}  
  
resource "aws_instance" "name" {  
  ami           = "ami-0dccc1e21636832c5d" →ami id of respective region  
  instance_type = "t2.micro" →configuration  
}
```

Save this

Now run the command → \$ **terraform init**

Terraform init initializes the working directory which consists of all the configuration files.

```
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!
```

Now run the command → **terraform fmt**

Used to automatically format Terraform configuration files to adhere to a consistent style. It helps maintain readability.

```

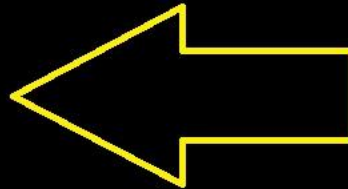
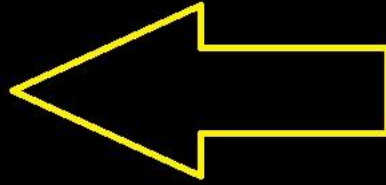
root@ip-172-31-15-149 ~]# cat main.tf
provider "aws" {
  region = "us-west-2"

  resource "ec2_instance" "example" {
    ami = "ami-0d593311db5abb72b"
    instance_type = "t2.micro"
    tags = {
      Name = "RAHAM-INSTANCE"
    }
  }
}

root@ip-172-31-15-149 ~]# terraform fmt
main.tf
root@ip-172-31-15-149 ~]# cat main.tf
provider "aws" {
  region = "us-west-2"

  resource "ec2_instance" "example" {
    ami          = "ami-0d593311db5abb72b"
    instance_type = "t2.micro"
    tags = {
      Name = "RAHAM-INSTANCE"
    }
  }
}

```



We can observe the difference before and after “fmt” in the above fig

Now run the command → **terraform plan**

Terraform plan is used to create an execution plan to reach a desired state of the infrastructure. Changes in the configuration files are done in order to achieve the desired state.

(Shows what it is going to do)

```

C:\Users\zealv\Desktop\kplabs-terraform>terraform plan

Terraform used the selected providers to generate the following execution plan. Resource actions are indicated with the following symbols:
+ create

Terraform will perform the following actions:

# aws_instance.myc2 will be created
+ resource "aws_instance" "myec2" {
  + ami          = "ami-00c39f71452c08778"    Provided by us
  + arn          = (known after apply)
  + associate_public_ip_address = (known after apply)
  + availability_zone = (known after apply)
  + cpu_core_count = (known after apply)
  + cpu_threads_per_core = (known after apply)
  + disable_api_stop = (known after apply)
  + disable_api_termination = (known after apply)
  + ebs_optimized = (known after apply)
  + get_password_data = false
  + host_id       = (known after apply)
  + host_resource_group_arn = (known after apply)
  + iam_instance_profile = (known after apply)
  + id           = (known after apply)
  + instance_initiated_shutdown_behavior = (known after apply)
  + instance_state = (known after apply)
  + instance_type = "t2.micro" provided by us
}

```

Now run the command → \$ **terraform apply**

Terraform apply then makes the changes in the infrastructure as defined in the plan, and the infrastructure comes to the desired state. {Deploy the infrastructure as per the code}
(Creates the instance)

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

Terraform used the selected providers to generate the following execution plan. Resource actions are indicated with the following symbols:
+ create

Terraform will perform the following actions:

# aws_instance.myc2 will be created
+ resource "aws_instance" "myec2" {
  + ami           = "ami-00c39f71452c08778"
  + ...           = (known after apply)
```

We need to confirm the creation action by entering yes (\$ terraform apply --auto-approve for auto approval without time waste)

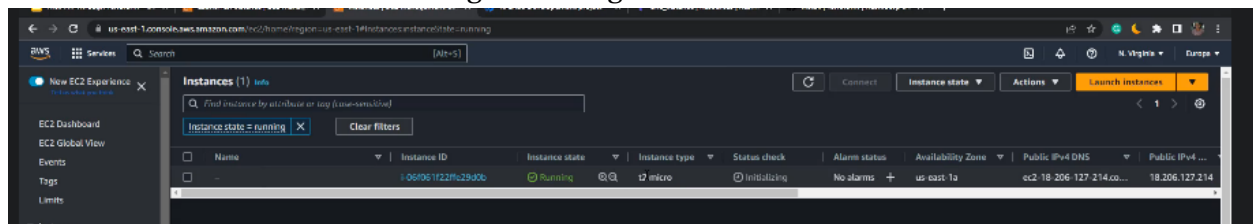
```
Do you want to perform these actions?
  Terraform will perform the actions described above.
  Only 'yes' will be accepted to approve.

  Enter a value: yes

aws_instance.myc2: Creating...
aws_instance.myc2: Still creating... [10s elapsed]
aws_instance.myc2: Still creating... [20s elapsed]
aws_instance.myc2: Still creating... [30s elapsed]
aws_instance.myc2: Creation complete after 37s [id=i-06f061f22ffe29d0b]

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

Now we can see the instance running in that region.



As there is no name to the created instance, we can give the name by changing the code.

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

resource "aws_instance" "desired_name" {
  ami           = "ami-0dccc1e21636832c5d" →ami id of respective region
  instance_type = "t2.micro" →configuration

  tags = {
    Name = "myfirstinstance"
  }
}
```

Save it and run the command →\$terraform apply.
(you can terraform plan first and then terraform apply later if you want)

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.
```

Saving Terraform Plan as a file:

We can use the optional `-out=FILE` option to save the generated plan to a file on disk, which we can later execute by passing the file to `terraform apply` as an extra argument.

Example: `$ terraform plan -out = demopath (filename)`

```
This plan was saved to: demopath
```

```
To perform exactly these actions, run the following command to apply:  
terraform apply "demopath"
```

Requiring providers

Each Terraform module must declare which providers it requires, so that Terraform can install and use them. Provider requirements are declared in a `required_providers` block.

A provider requirement consists of a local name, a source location, and a version constraint:

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

The `required_providers` block must be nested inside the top-level terraform block (which can also contain other settings).

There is also an existing provider with the source address `hashicorp/terraform`, which is an older version of the now-built-in provider that was used by older versions of

Terraform.hashicorp/terraform is not compatible with Terraform v0.11 or later and should never be declared in a required_providers block.

Creating Github Repository using Terraform

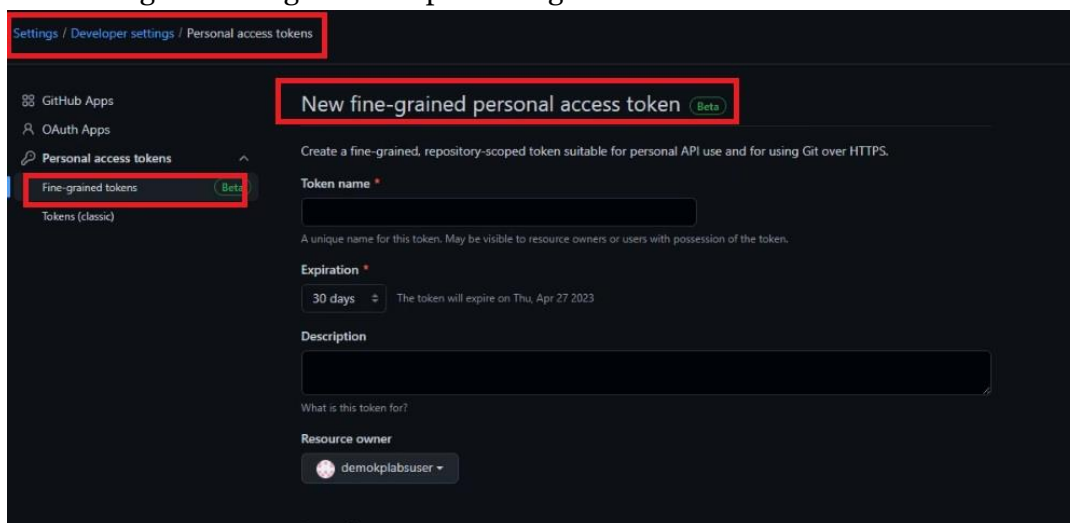
Get the code format for github provider and edit it.

<https://registry.terraform.io/providers/integrations/github/latest/docs>

Also, Personal access token also needed for connection between terraform and github.

Generating Personal access token

In Github go to setting → developer settings → Personal Access tokens



Generate fine grained token, by selecting access to the repos and required permissions.

```
terraform {
  required_providers {
    github = {
      source = "integrations/github" #making github as provider
      version = "~> 5.0"
    }
  }
}

# Configure the GitHub Provider #giving access to terrafrom with token details

provider "github" {
  token = "XXXXXXXXXXXX"
  owner = "devopsschool-demo-temporary"
}

resource "github_repository" "github-repo-1" {#creating repo with simple config
  name = "terraform-demo-1"
```

```

        description = "My awesome codebase"

        visibility = "public"
    }

```

Save it with .tf extension

Run the commands

\$ terraform init - Download Providers & Modules

\$ terraform plan - DRY RUN

\$ terraform apply - Create a resources &&& store a response IN STATE file(json)

After practice \$ terraform destroy - Destroy a resource using STATE file(json) **type yes for confirmation** (\$ terraform destroy -auto-approve for auto approval without time waste)

Terraform destroy allows us to destroy **all the resources** that are created within the folder.

Terraform destroy with **-target** flag allows us to destroy specific resource.

Target flag s combination of resource type.local resource name

Ex: \$ terraform destroy -target github_repository.github-repo-1 deletes the github repository only.

Or else we comment out the code which is not required /* code */

To create a branch of existing repo

```

terraform {
  required_providers {
    github = {
      source = "integrations/github" #making github as provider
      version = "~> 5.0"
    }
  }
}

provider "github" {

  token = "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"
}

resource "github_repository" "terra_repo" {
  name = "terra_repo"
  visibility = public
}

```

```
resource "github_branch" "feature" {
  repository = "reponame"
  branch = "feature"
  source_branch = "main"
}
```

-----*****-----

Terraform Variables

Terraform input variables are used as parameters to input values at run time to customize our deployments. Input terraform variables **can be defined in the main.tf configuration** file but it is a **best practice** to define them in a **separate variable .tf** file to provide better readability and organization.

A variable is defined by using a variable block with a label. The label is the name of the variable and must be unique among all the variables in the same configuration

Variables → Repeated static values within the code

Variable Assignment

Different approaches to variable assignment in Terraform are

1. Variable Defaults
2. Command-line flags
3. From a file
4. Environment variables

Variable default approach:

In this approach, we create a variable.tf file and define a variable and assign a default value to a variable.

If no explicit value is defined to the variable, then default value is taken into consideration.

For example, in place of instance type we write a variable name

```
provider "aws" {
  region = "us-west-2"
}
resource "aws_instance" "name" {
  ami           = "ami-0dccc1e21636832c5d"
  instance_type = "var.instancetype"
}
```

Now we create a file **variable.tf** with below simple code

```
variable "instancetype" {
  default = "t2.micro"
```

```
}
```

Or

```
variable "instancetype" {  
  description = "this is variable block"  
  type = "string"  
  default = "t2.micro"  
}
```

Now if we run terraform plan for ec2_variable.tf , we can see that instance type is t2.micro

```
+ 1d = (known after apply)  
+ instance_state = (known after apply)  
+ instance_type = "t2.micro"  
+ ipv6_address_count = (known after apply)  
+ ipv6_addresses = (known after apply)  
+ key_name = (known after apply)
```

////////////////////////////////////

What will happen if we don't provide a default value?

Let **variable.tf** contains only variable "instancetype" { }

we run terraform plan then it asks to enter a value as no default value is specified

```
C:\Users\Zeal Vora\Desktop\terraform\terraform variables>terraform plan  
var.instancetype  
Enter a value: t2.medium
```

2. Command-line flags

In the similar example above if we have a variable and we have not provided a default value or we want to override the value. We can do so by providing value in the command line.

```
terraform plan --var="instance_type=t2.small"
```

Now we run \$terraform plan -var ="instancetype=t2.small" (here specific explicit value is specified, so the default instance type is not taken into consideration)

```
C:\Users\Zeal Vora\Desktop\terraform\terraform variables>terraform plan -var="instancetype=t2.small"
```

```
+ id = (known after a
+ instance_state = (known after a
+ instance_type = "t2.small"
+ ipv6_address_count = (known after a
+ ipv6_addresses = (known after a
```

3. From a file

Another way you can provide value is using file. Create a **terraform.tfvars** file and provide value to variables.

Let **variable.tf** contains only variable "instancetype" { }

```
//terraform.tfvars instance_type="t2.micro"
```

Now when you do terraform plan the instance type value will be taken from **terraform.tfvars** file.

Note: File naming is important here. Terraform by default only looks for terraform.tfvars file.

Let's say for some reason you want to use a custom file name. Create a new file **custom.tfvars** and delete the **terraform.tfvars**

Now in order to use custom.tfvars you can provide the file name in CLI like below.

```
terraform plan -var-file="custom.tfvars"
```

```
Command Prompt
C:\Users\Zeal Vora\Desktop\terraform\terraform variables>terraform plan -var-file="custom.tfvars"
```

4. Environment variables

Let's set an environment variable:

In windows:

```
setx TF_VAR_instance_type t2.micro
```

The TF_VAR_<VARIABLE> <Value> is terraform specific.

```
C:\Users\Zeal Vora\Desktop\terraform\terraform variables>setx TF_VAR_instancetype m5.large
SUCCESS: Specified value was saved.
```

Once value is saved, though you run terraform plan it doesn't work. We need to open new cmd

```
C:\Users\Zeal Vora\Desktop\terraform\terraform variables>echo %TF_VAR_instancetype%
%TF_VAR_instancetype%      No value shown in old cmd

C:\Users\Zeal Vora\Desktop\terraform\terraform variables>
Microsoft Windows [Version 10.0.18363.778]
(c) 2019 Microsoft Corporation. All rights reserved.

C:\Users\Zeal Vora>echo %TF_VAR_instancetype%
t5.large                    assigned value shown in cmd

C:\Users\Zeal Vora>
```

So run terraform plan in new cmd

In Linux/Mac

export TF_VAR_instancetype="t2.nano"

then run terraform plan

*****-----*****

Variable Declaration:

You declare variables using the variable block in your Terraform configuration files. You specify the variable name and optionally provide a default value and a description.

In Terraform, variables can have different types, allowing you to specify constraints on the data that can be passed into your Terraform modules.

The variable can be referred to by using the var keyword followed by . and then the <variable_name>.

Ex: var.vpc_name

Primitive Variable Types

Below are the primitive variable types supported by Terraform:

- ✓ String
- ✓ Number
- ✓ Bool or Boolean

String: String variable type can hold any text-based value. For example, string type can be used to represent a region, such as "us-west-1" or "eu-central-1".

Declaring string variable

```
variable "region" {
  type    = string
  description = "The AWS region where resources will be provisioned"
  default  = "us-west-2"
}
```


Using string variable

```
resource "aws_instance" "example" {  
  ami      = "ami-0c55b159cbfaffe1fo"  
  instance_type = "t2.micro"  
  availability_zone = "${var.region}"  
}
```

Number

The number variable type can be an integer or a floating-point value such as 2.5, etc. We can, for example, define the `instance_count` variables to represent the number of instances deployed.

Declaring number variable

```
variable "instance_count" {  
  type    = number  
  description = "The number of instances to launch"  
  default  = 2  
}
```

Using number variable

```
resource "aws_instance" "example" {  
  count      = var.instance_count  
  ami       = "ami-0c55b159cbfaffe1fo"  
  instance_type = "t2.micro"  
}
```

Boolean

A Boolean value, either true or false. bool values can be used in conditional logic.

Declaring boolean variable

```
variable "enable_monitoring" {  
  type    = bool  
  description = "Enable detailed monitoring for EC2 instances"  
  default  = true  
}
```

Using Boolean variable

```
resource "aws_instance" "example" {  
  ami      = "ami-0c55b159cbfaffe1fo"  
  instance_type = "t2.micro"  
  monitoring = var.enable_monitoring  
}
```

The Terraform Variable Type Boolean supports another approach to specify “true” or “false.” We can specify **zero “0”** with double quotes, and the Terraform will automatically convert it to false. The same idea happens for “true,” we can specify **“1,”** and it will be converted to “true.

- ❖ What if we want to change the instance type just for the xxxxxxxx environment
- ❖ We can assign a value to the variable through the command line.
- ❖ terraform plan -var="instance_type=t2.large" or
- ❖ TF_VAR_instance_type="t2.small" terraform plan

Complex Variable Types

Below are the complex variable types supported by Terraform:

- ✓ List
- ✓ Map
- ✓ Object

List

Type is the List. It's a sequence of values. And each value contains one **index**. And the first element index of this List is zero.

Declaring list variable:

```
variable "security_groups" {  
  type      = list(string)  
  description = "List of security group IDs"  
  default    = ["sg-12345678", "sg-87654321"]  
}
```

Using list variable:

```
resource "aws_instance" "example" {  
  ami          = "ami-0c55b159cbfafa1fo"  
  instance_type = "t2.micro"  
  security_groups = var.security_groups[count.index]  
}
```

In the above example the variable **security_groups[0]** contains the value "sg-12345678", variable **security_groups[1]** contains the value "sg-87654321"

Map

The map variable type is used to create variables holding key-value pair pairs. This allows you to add a list of named values in your Terraform configuration. Each value will be associated with a unique Key.

Declaring list variable:

```
variable "tags" {  
  type      = map(string)  
  description = "Map of resource tags"  
  default   = {  
    Name      = "ExampleInstance"  
    Environment = "Production"  
  }  
}
```

Using list variable:

```
resource "aws_instance" "example" {  
  ami          = "ami-0c55b159cbfaffe1fo"  
  instance_type = "t2.micro"  
  tags         = var.tags  
}
```

In Terraform, there isn't a direct "object" variable type like you might find in other programming languages. However, you can achieve similar functionality by using a map variable with nested structures to represent objects.

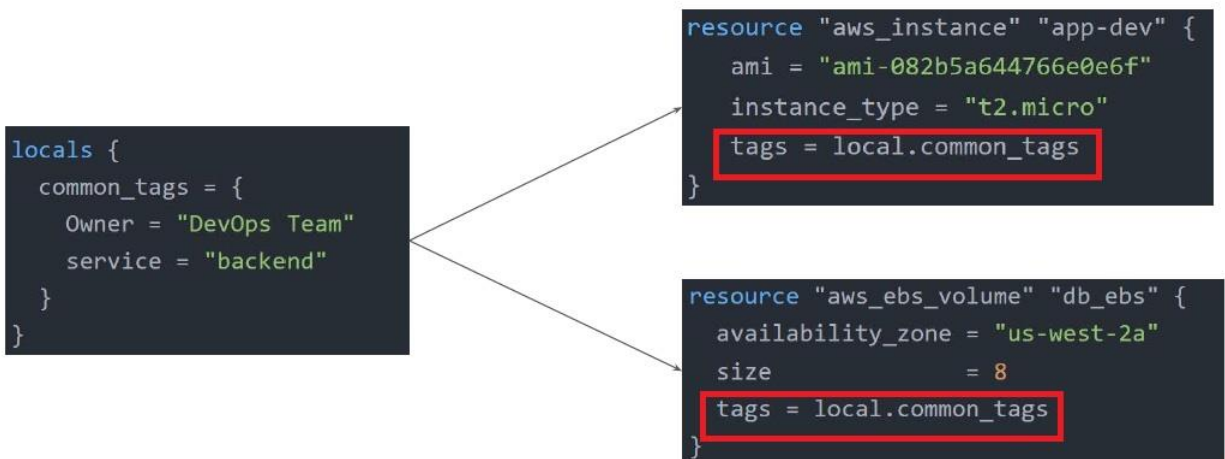
Comments in Terraform

The Terraform language supports three different syntaxes for comments

Type	Description
#	begins a single-line comment, ending at the end of the line.
//	also begins a single-line comment, as an alternative to #.
/* and */	are start and end delimiters for a comment that might span over multiple lines.

Local Values

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



In the above example the resources created will get inherit the key value pair defined in the local. (since local values are specified as tags in the resources)

Terraform locals are quite similar to terraform variables but Terraform locals do not change their value. On the other hand, if you talk about Terraform input variables then it is dependent on user input and it can change its value. So, if you have a very large Terraform file where you need to use the same values or expressions multiple times then Terraform local can be useful for you.

Output values // Output variables

In Terraform, output values allow you to extract information from your infrastructure after it has been provisioned. These outputs can be useful for **displaying important information to users**, sharing data between different Terraform configurations, or integrating with other tools and systems. Outputs are typically defined in a Terraform configuration file using the output block.

Here's how you can define output values in a Terraform configuration file: (before writing this you should have code to create instance. So, that we can display/print its related information)

```
output "instance_ip" {
  value = aws_instance.ourinstance_name.public_ip
}

output "instance_id" {
  value = aws_instance.ourinstance_name.id
}
```

In this example:

instance_ip is the name of the output value.
aws_instance. ourinstance_name public_ip is the attribute of the AWS EC2 instance resource (aws_instance. ourinstance_name) that we want to expose as an output value.

After running the command terraform apply, terraform will display the output values:

Outputs:

```
instance_ip = 203.0.113.10 #for example  
instance_id = i-0123456789abcdef0 # for example
```

Note: Now if you want to hide the sensitive info (like IP) use the key called sensitive.

```
output "instance_ip" {  
  value = aws_instance.ourinstance_name.public_ip  
  sensitive =true  
}
```

Now after running terraform apply, we get output as
instance_ip = <sensitive>

.tfvars files

When we want use different set of variables with the same main.tf file we go for .tfvars files.

Suppose we can have different variables for prod and dev. Then we can create respective variable files as prod.tfvars and dev.tfvars.

We can use these variables at the time of terraform plan as

➔ terraform plan -var-file=prod.tfvars So that infra creation will be done for prod

➔ terraform plan -var-file=dev.tfvars so that infra creation will be done for dev

Note: when we run terraform apply terraform apply with the second set of variables, the previous configuration will be overwritten and state file will also get changed

Example:

main.tf

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

```
variable "instance_type" {}  
variable "ami" {}  
variable "count" {}
```

```
resource "aws_instance" "example" {  
  count      = var.count  
  ami       = var.ami  
  instance_type = var.instance_type  
  
  # Other resource configurations...  
}
```

dev.tfvars

```
instance_type = "t2.micro"  
ami          = "ami-0c55b159cbf1f0"  
count        = 3
```

prod.tfvars

```
instance_type = "t2.large"  
ami          = "ami-0123456789abcdefo"  
count        = 5
```

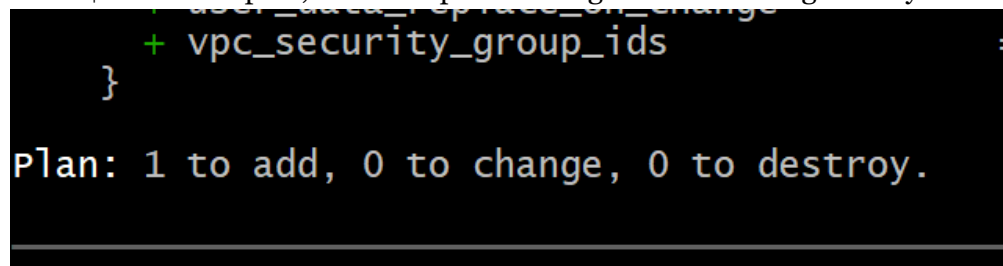
Terraform state file

When you run **terraform apply** command to create an infrastructure on cloud, terraform creates a **state file** called “**terraform.tfstate**”. This State File contains **full details** of resources in our terraform code. When you modify something on your code and apply it on cloud, terraform will look into the **state file**, and compare the changes made in the code from that state file and the changes to the infrastructure based on the state file. Terraform State File is written in a simple readable language “**JSON**”

State file keeps track of the resources declared in your Terraform configuration files (.tf) and their current state in the real-world infrastructure. The state file allows Terraform to understand what resources it manages, their current configuration, and any dependencies between them.*

For example, we have two EC2 instances (resources) within one main.tf file. And we destroyed one ec2 instance with `terraform destroy -target=aws_instance.giveninstancename[index]`. Terraform knows that ec2 instance is destroyed as it is in terraform.tfstate the ec2 instance resource is still present.

Later if we run `$ terraform plan`, it tries to plan a change that is creating destroyed ec2 instance.

A screenshot of a terminal window with a dark background. It shows a snippet of Terraform configuration code: `+ vpc_security_group_ids` followed by a closing brace `}`. Below the code, the output of the `terraform plan` command is displayed: `Plan: 1 to add, 0 to change, 0 to destroy.`

Thus, though destroyed we can create with plan and apply commands again because of state file.

Note: Never try to edit terraform.tfstate unnecessarily.

Desired State and Current State

- In Terraform, the desired state is the state that you want your infrastructure to be as defined in your Terraform Configuration files.
- The current state is the actual state of the infrastructure as represented in the Terraform state file.
- When you run the Terraform apply, terraform compares the desired state with the current state and makes changes as needed to bring the infrastructure into the desired state.

Terraform tries to ensure that the deployed infrastructure is based on the desired state with the help of state file. If there is a difference between the desired and current state, terraform plans a description of the necessary changes to achieve the desired state.

What is the Terraform Backup State File?

Terraform automatically creates a backup file called **terraform.tfstate.backup** when you run certain Terraform commands like terraform apply or terraform destroy. This backup file is a copy of the **previous state file (terraform.tfstate)** before any changes are applied.

The purpose of this backup file is to provide a safety net in case anything goes wrong during the execution of Terraform commands. If the state file becomes corrupted or if changes are accidentally applied that cause issues, you can use the backup file to restore the previous state.

Whenever we modify something on our code and apply it, our tfstate file will change our resources' information. At that point this backup file acts as an old version of the state file. So the modified resources details are in the tfstate file, and the old tfstate file will be transferred to tfstate.backup file.

-----*****-----

Backend and locking

As the state file contains all the configuration details, it is not safe to have state file in our local machine. It is generally not recommended to commit .tfstate files to Git or any other version control system. .tfstate files contain sensitive information about the infrastructure managed by Terraform, including the current state of resources and the configurations used to create them. If these files are committed to version control and made publicly available, they could potentially be accessed by unauthorized users, posing a security risk.

Few problems if we commit a state file to a repository:

Concurrency: If 2 or more developers are working in the stack, they won't see the other state until it's pushed to the repository

Automated Deployment: A CI tool that deploys the stack automatically would need to commit the new state file to the repository

Easily corruptible: In case of a merge conflict or human error, the state file can be corrupted or gone.

Instead of having a local JSON file holding the state, the state file is uploaded to an **S3 bucket**.

And when two or more users work on the infrastructure simultaneously, problems may arise in the creation of the resources, as there will be a situation such as executing another process before the state is finalized. Therefore, terraform can **lock your state** to prevent other users from breaking our infrastructure using the same state file at the same time.

AWS S3 bucket supports this lock feature with the Amazon DynamoDB table.

For this in the main file along with resources you need, create a S3 bucket and Dynamo-Db table

Example main.tf

```
provider "aws" {
  region = "us-east-1"
}
resource "aws_instance" "myec2" {
  ami = "ami-0c55b159cbfaffe1fo"
  instance_type = "t2.micro"
}

#creating S3 bucket

resource "aws_s3_bucket" "mystatebucket" {
  bucket = "state_file_bucket"
}

#creating dynamo db table

resource "aws_dynamodb_table" "terra_lock" {
  name = "terraform_lock"
  billing_mode = "PAY_PER_REQUEST"
  hash_key = "LockID"

  attribute {
    name = "LockID"
    type = "S"
  }
}
```

Now after running terraform apply create backend and run that file too

backend.tf

```
terraform {
  backend "s3" {
```

```

bucket = "state_file_bucket"
region = "us-east-1"
key = "beacreful/terraform.tfstate"
#key is nothing path in S3 bucket where you want to keep the file
dynamodb_table = "terraform_lock"
}
}

```

-----*****-----

The terraform -f unlock command is intended for local state files only and does not work with remote backends such as S3 and DynamoDB.

Using Terraform CLI: Terraform provides a terraform force-unlock command that allows you to manually release a lock on a state file held by a particular lock ID. However, this command should be used with caution, as forcibly unlocking a state file can lead to corruption or inconsistencies if multiple Terraform operations are running concurrently.

terraform force-unlock LOCK_ID

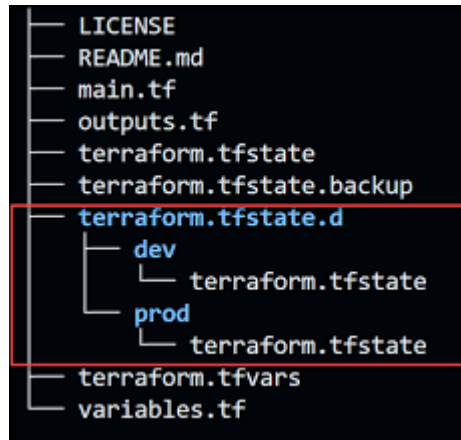
Workspace

A Terraform workspace is a collection of infrastructure that Terraform manages using workspaces instead of directories. A workspace contains everything Terraform needs to manage a given collection of infrastructure.

Terraform workspaces allow you to manage multiple environments (such as development, staging, and production) using a single configuration. Each workspace maintains its own state, allowing you to have different infrastructure for each environment without duplicating your configuration files.

{{{why can't we use one main.tf file and multiple. tfvars ?? because we have all the main.tf and. tfvars in single directory. When we run terraform apply first say with dev.tfvars it creates a state file with dev conf. Next when we run terraform apply with qa.tfvars the state file will be over written and can't keep track of both the environments or can't have different state files. In such a case we have workspaces available}}}

Each workspace in Terraform typically has its own state file to manage resources independently. The terraform.tfstate.d directory gets created when we create a workspace and is used to store these state files, possibly in a structured way to differentiate between different workspaces.



For example:

Create main.tf

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

```
Resource "aws_instance" "test"
{
  ami = "ami-0c55b159cbfaffe1fo"
  instance_type = var.instance_type
}
```

Now I want different instance types for different environments. Let's say for dev 't2.micro', for qa 't2.medium', for prod 't2.large'

1st Way—create three .tfvars in the same directory where main.tf is created and when running terraform apply need to give the respective .tfvars like terraform apply -var-file=dev.tfvars.

2nd way – create .tfvars in respective environment and run terraform apply directly. So, that main.tf takes the variable file in the respective directory/environment.

Suppose if we run → **terraform workspace list** as no custom workspaces have been created, it shows output as *default

Now create a workspace dev ,qa, prod

→ **terraform workspace new dev**

→ terraform workspace new qa

→ terraform workspace new prod

Go to dev environment

→

terraform workspace select dev

create dev.tfvars

```
variable "instancetype" {
description = "this dev instace type"
default = "t2.micro"
}
```

Now run terraform init, plan, and apply. Instance with t2.micro will be created for dev purpose

Similarly select the other environments and create respective .tfvar file and run terraform apply

terraform workspace show → Displays information about the currently selected workspace.
terraform workspace delete workspace_name → Deletes the workspace with the specified name.
This command will not work if you are currently in the workspace you are trying to delete.
Also in this scenario, instead of having multiple .tfvars files as it we have only variable i.e. instance type we can declare this within main.tf file like

```
variable "instancetype" {
type = map(string)

default = {

"dev" = "t2.micro"
"qa" = "t2.medium"
"prod" = "t2.large"
}
}
```

Exercise: With Workspace concept use S3 as backend and see how state files are saved

```
terraform {
  backend "s3" {
    bucket = "example-bucket"
    key    = "terraform/${terraform.workspace}.tfstate"
    region = "us-west-2"
  }
}
```

When have multiple workspaces and multiple state files within a single S3 bucket (backend) keep each state file in different in different directory.

```
terraform-state-bucket/
├── dev/
│   └── terraform.tfstate
├── qa/
│   └── terraform.tfstate
└── prod/
    └── terraform.tfstate
```

Terraform code to copy local files to S3 bucket

```
provider "aws" {  
  region = "us-west-2" # Update with your desired AWS region  
}  
  
resource "aws_s3_bucket" "example_bucket" {  
  bucket = "example-bucket"  
  acl    = "private"  
}  
  
resource "aws_s3_bucket_object" "example_object" {  
  bucket = aws_s3_bucket.example_bucket.bucket  
  key    = "example_file.txt" # Specify the name/key of the file in the bucket  
  source = "/path/to/local/file/example_file.txt" # Path to the local file to upload  
  acl    = "private"  
}
```

Modules

A Terraform module is a set of Terraform configuration files in a single directory. Even a simple configuration consisting of a single directory with one or more .tf files is a module. When you run Terraform commands directly from such a directory, it is considered the root module.

We can have different module directories like module-1 for one instance and module-2 for another instance with different configurations.

Or we can have different modules like one module for network, one module for storage, one module for instance etc.

In a module directory we can have a main.tf (child main.tf), variable.tf, output.tf. These modules are called in a main.tf(parent) which is outside of the all the modules.

Once you define the modules the next important step would be to **call the modules** from the parent main.tf file and terraform makes it easy to call the modules **based on the relative paths**.



Provisioner

In Terraform, provisioners are used to run commands or scripts on local or remote machines. They can also transfer files from a local environment to a remote one. They allow you to perform tasks such as initializing, configuring, or bootstrapping resources after they've been created or before they're destroyed.

Local-exec Provisioner: The local-exec provisioner executes commands or scripts on the machine where Terraform is running (the local machine). This is typically used for tasks that need to be performed locally, such as initializing or configuring resources on the Terraform host.

Ex:

```
resource "aws_instance" "example" {  
  # Instance configuration...  
  
  provisioner "local-exec" {  
    command = "echo 'Instance created'"  
  }  
}
```

Remote-exec Provisioner: The remote-exec provisioner executes commands or scripts on a remote machine (the target resource). This is useful for tasks like installing software, configuring services, or performing other actions on the resource after it has been created.

Ex:

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

  provisioner "remote-exec" {
    inline = [
      "sudo apt-get update",
      "sudo apt-get install -y nginx",
    ]
  }
}
```

File Provisioner: The file provisioner is used to copy files or directories from the machine where Terraform is executed (local machine) to a target resource (remote machine).

Ex:

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

  provisioner "file" {
    source      = "local/path/to/file.txt"
    destination = "/remote/path/to/destination.txt"
  }
}
```

-----exercise-----

Creating Security group and editing rule

https://registry.terraform.io/providers/hashicorp/aws/latest/docs/resources/security_group

elastic_ip

<https://registry.terraform.io/providers/hashicorp/aws/latest/docs/resources/eip>

```
resource "aws_instance" "example" {
  ami          = "ami-12345678"
  instance_type = "t2.micro"
  subnet_id    = aws_subnet.example.id

  # Reference the security group ID from another resource
  vpc_security_group_ids = [aws_security_group.example.id]

  tags = {
    Name = "ExampleInstance"
  }
}
```

```

}

data "aws_security_group" "example" {
  name = "example-security-group"
}

```

This is in a separate file or module

-----exercise-----

Cross resource attribute reference

In Terraform, a cross-resource attribute typically refers to an attribute of one resource that is used within the configuration of another resource. This is usually achieved by directly referencing the attributes or outputs of one resource within the configuration of another resource.

security_group.tf

```

resource "aws_security_group" "example" {
  name      = "example-security-group"
  description = "Example security group for EC2 instance"

  # Define your security group rules here
  # For example:
  ingress {
    from_port = 22
    to_port   = 22
    protocol  = "tcp"
    cidr_blocks = ["0.0.0.0/0"]
  }

  egress {
    from_port = 0
    to_port   = 0
    protocol  = "-1"
    cidr_blocks = ["0.0.0.0/0"]
  }
}

output "security_group_id" {
  value = aws_security_group.example.id
}

```

ec2_instance.tf


```
resource "aws_instance" "example" {
  ami          = "ami-12345678"
  instance_type = "t2.micro"
  subnet_id    = aws_subnet.example.id

  # Reference the security group ID from the output of the security group resource
  vpc_security_group_ids = [security_group.security_group_id]
                        or
  vpc_security_group_ids = "${aws_security_group.example.id}"

  # string interpolation--${...} indicates that terraform will replace the expression inside
  # the curly braces with its calculated value.

  tags = {
    Name = "ExampleInstance"local exe
  }
}
```

Terraform code for creating a VPC and respective subnet with RT, IGW to have an Ec2 instance to deploy an app

```
# Define the AWS provider configuration.
provider "aws" {
  region = "us-east-1" # Replace with your desired AWS region.
}

variable "cidr" {
  default = "10.0.0.0/16"
}

resource "aws_key_pair" "example" {
  key_name   = "terraform-demo-abhi" # Replace with your desired key name
  public_key = file("~/ssh/id_rsa.pub") # Replace with the path to your public key file
}

resource "aws_vpc" "myvpc" {
  cidr_block = var.cidr
}

resource "aws_subnet" "sub1" {
  vpc_id            = aws_vpc.myvpc.id
  cidr_block        = "10.0.0.0/24"
  availability_zone = "us-east-1a"
  map_public_ip_on_launch = true
}

resource "aws_internet_gateway" "igw" {
  vpc_id = aws_vpc.myvpc.id
}

resource "aws_route_table" "RT" {
  vpc_id = aws_vpc.myvpc.id

  route {
    cidr_block = "0.0.0.0/0"
    gateway_id = aws_internet_gateway.igw.id
  }
}

resource "aws_route_table_association" "rta1" {
  subnet_id   = aws_subnet.sub1.id
  route_table_id = aws_route_table.RT.id
}

resource "aws_security_group" "webSg" {
```

```

name = "web"
vpc_id = aws_vpc.myvpc.id

ingress {
  description = "HTTP from VPC"
  from_port   = 80
  to_port     = 80
  protocol    = "tcp"
  cidr_blocks = ["0.0.0.0/0"]
}
ingress {
  description = "SSH"
  from_port   = 22
  to_port     = 22
  protocol    = "tcp"
  cidr_blocks = ["0.0.0.0/0"]
}

egress {
  from_port = 0
  to_port   = 0
  protocol  = "-1"
  cidr_blocks = ["0.0.0.0/0"]
}

tags = {
  Name = "Web-sg"
}
}

resource "aws_instance" "server" {
  ami           = "ami-0261755bbcb8c4a84"
  instance_type = "t2.micro"
  key_name      = aws_key_pair.example.key_name
  vpc_security_group_ids = [aws_security_group.webSg.id]
  subnet_id     = aws_subnet.sub1.id

  connection {
    type     = "ssh"
    user     = "ubuntu" # Replace with the appropriate username for your EC2 instance
    private_key = file("~/ssh/id_rsa") # Replace with the path to your private key
    host     = self.public_ip
  }

  # File provisioner to copy a file from local to the remote EC2 instance
  provisioner "file" {
    source = "app.py" # Replace with the path to your local file
  }
}

```

```

    destination = "/home/ubuntu/app.py" # Replace with the path on the remote
instance
}

provisioner "remote-exec" {
  inline = [
    "echo 'Hello from the remote instance'",
    "sudo apt update -y", # Update package lists (for ubuntu)
    "sudo apt-get install -y python3-pip", # Example package installation
    "cd /home/ubuntu",
    "sudo pip3 install flask",
    "sudo python3 app.py &",
  ]
}
}

```

Data Resources

Data sources in Terraform are used to get information about resources external to Terraform, and use them to set up your Terraform resources. This information is typically retrieved from external systems or services such as cloud providers (AWS, Azure, GCP), APIs, databases, DNS servers, etc.

Simply data sources can query external sources and return the data. Performs read only operation.

Some examples of data sources include:

- Machine image IDs from a cloud provider
- Terraform outputs from other configurations
- A list of IP addresses a cloud provider exposes

Data sources allow Terraform to use information defined outside of Terraform, defined by another separate Terraform configuration, or modified by functions.

Data blocks are used to define and configure data sources. They are similar to resource blocks, but instead of creating a resource, they define a data source.

syntax for data source in aws terraform

```

data "aws_<RESOURCE_TYPE>" "<RESOURCE_NAME>" {
  # Arguments and filters to specify the resource you want to retrieve
  # For example:
  # - filters
  # - identifiers
  # - optional configuration attributes

```

```
}
```

Ex:

```
data "aws_instance" "myawsinstance" {
  filter {
    name = "tag:Name"
    values = ["Terraform EC2"]
  }

  depends_on = [
    "aws_instance.ec2_example"
  ]
}
```

Filters: They are used within data source blocks to retrieve only the resources that meet the specified conditions. Filters allow you to target resources based on various attributes such as tags, names, IDs, states, and more, depending on the resource type.

depends_on: The second important parameter is depends_on because data source does not know by its own which resource it belongs to, so we are going to add the depends_on parameter.

Import

The import command is used to import existing infrastructure into your Terraform state. This is useful when you have infrastructure that was created **outside of Terraform**, and you want to manage it using Terraform going forward.

Before importing the resource, we should create its configuration in the root module(main.tf) and then we can import, see in the state file.

Ex: we have created an EC2 instance manually through console. Now if we want to import the created infrastructure details, first you should have similar config with its attribute

Syntax: terraform import RESOURCE_TYPE.NAME ID

```
provider "aws" {
  region = "ap-south-1"
}
```

#creating config for instance which is created manually

```
resource "aws_instance" "imp_instance" {  
  ami = "give the ami of manually configured instance" copy from there  
  instance_type = "give that of manually created "  
}
```

Save this file. Now you can import by using the command.

→ terraform import aws_instance. imp_instance id_of_created_instance

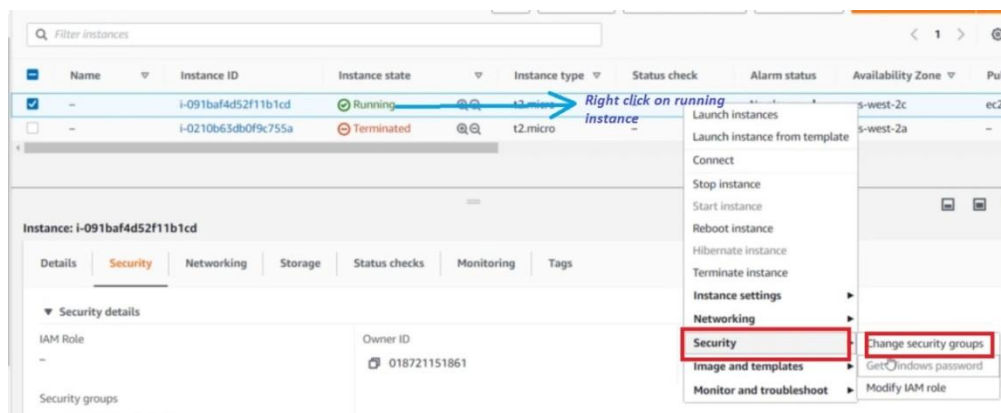
Now that manually created instance infrastructure data is imported with the resource name imp_instance

Terraform Refresh

Terraform can create infrastructure-based configuration we specified. There are chances that the infrastructure gets modified manually. The terraform refresh command will check the latest of your infrastructure and update the state file accordingly.

Note: We shouldn't typically use this command, because Terraform automatically performs the same refreshing actions as a part of creating a plan.

*** Suppose we changed the security group of a running instance from default to custom



Change security groups Info

Amazon EC2 evaluates all the rules of the selected security groups to control inbound and outbound traffic to and from your instance. You can use this window to add and remove security groups.

Instance details

Instance ID i-091baf4d52f11b1cd	Network interface ID eni-0b0d4c348c379e518
------------------------------------	---

Associated security groups
Add one or more security groups to the network interface. You can also remove security groups.

Security groups associated with the network interface (eni-0b0d4c348c379e518)

Security group name	Security group ID	
custom	sg-0fb2ffa89cdf5d271	<input type="button" value="Remove"/>

Now we run the command \$ **terraform refresh** the security group in the configuration file changes from default to custom by modifying the **terraform state file**. *So, be watchful in using **terraform refresh** manually.*

Now though we run \$ terraform plan, it will not show/suggest any changes.

```
C:\Users\Zeal Vora\Desktop\kplabs-terraform>terraform plan
github_repository.example: Refreshing state... [id=terraform-repo]
aws_instance.myec2: Refreshing state... [id=i-091baf4d52f11b1cd]

No changes. Infrastructure is up-to-date.

This means that Terraform did not detect any differences between your
configuration and real physical resources that exist. As a result, no
actions need to be performed.
```

-----*****-----

Loop

<https://www.bogotobogo.com/DevOps/Terraform/Terraform-Introduction-AWS-loops.php>

[Using Loops in Terraform Code. Terraform is an infrastructure as code... | by Vartika Srivastava | The CloudTechner Blog](#)

Terraform has two main loop constructs: "for_each" and "count".

for_each

Used when resources change between different instances. It allows you to iterate over a list or map and create multiple instances of a resource or module based on the elements of that list or map. It also provides the flexibility of dynamically setting the attributes of each resource instance created.

The `for_each` meta-argument allows you to create multiple instances of a resource based on a map or set of strings. This is particularly useful when you want more control over the instances being created.

```
-----  
provider "aws" {  
  region = "ap-south-1"  
}  
  
variable "mytags" {  
  type = set(string)  
  default = ["dev", "qa", "prod"]  
}  
  
resource "aws_instance" "aadi" {  
  for_each = var.mytags  
  
  ami = "ami-0e670eb768a5fc3d4"  
  instance_type = "t2.micro"  
  
  tags = {  
    name = each.key  
  }  
}
```

```
-----  
variable "mytags" {  
  default = {  
  
    tag1 = {  
      name = "dev"  
    }  
    tag2 = {  
      name = "qa"  
    }  
    tag3 = {  
      name = "prod"  
    }  
  }  
}
```



```

}

resource "aws_instance" "aadi" {

  for_each = var.mytags

  ami = "ami-0e670eb768a5fc3d4"
  instance_type = "t2.micro"

  tags = {
    name = each.value.name
  }
}

-----

provider "aws" {

  region = "ap-south-1"

}

locals {
  mytags = {

    tag1 = "dev"
    tag2 = "qa"
    tag3 = "prod"
  }
}

resource "aws_instance" "aadi" {

  for_each = local.mytags

  ami = "ami-0e670eb768a5fc3d4"
  instance_type = "t2.micro"

  tags = {

    name = each.key
  }
}

```

```
}
```

count

Used when you are provisioning multiple resources that are identical or near identical. It allows you to loop over resources and modules.

The count meta-argument allows you to create multiple instances of a resource based on an integer value.

Count

The count argument is used to determine the number of instances to create for a particular resource. The count argument can be used in both a module as well as every resource type.

For example, instead of defining three virtual machines all with their own separate resource blocks, you could define one and add the count = 3 'meta-argument' into the resource block.

Creating multiple resources using the count argument #

```
resource "google_redis_instance" "example" {  
  count      = 3  
  name       = "redis-instance-${count.index}"  
  memory_size_gb = 10  
}
```

Understanding Challenge with Count

With the below code, terraform will create 3 instances. But the problem is that all will have the **same name**.

```
provider "aws" {  
  region = "us-west-2"  
}  
  
resource "aws_instance" "desired_name" {  
  ami           = "ami-odcc1e21636832c5d"  
  instance_type = var.instance_type
```

count = 3

```
  tags = {  
    name = "myec2"  
  }  
}
```

```
variable instance_type {  
  default = "t2.micro"
```

```
}
```

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

```
provider "aws" {
  region = "us-west-2"
}
resource "aws_instance" "desired_name" {
  ami          = "ami-odcc1e21636832c5d"
  instance_type = var.instance_type
```

count = 3

```
tags = {
  name = "myec2.${count.index}"
}
```

```
variable instance_type {
  default = "t2.micro"
}
```

3 instances will be created with names myec2.0, myec2.1, myec2.2

Understanding Challenge with Default Count Index

Having a username like myec2.0, myec2.1, myec2.2 might not always be suitable. Better names like dev-server, qa-server, prod-server is better. `count.index` can help in such scenario as well.

```
provider "aws" {
  region = "ap-south-1"
}
```

```
resource "aws_instance" "name" {
  ami          = "ami-06b72b3b2a773be2b"
  instance_type = var.instance_type
  count        = 3
```

```
tags = {
  name = var.instance_name[count.index]
}
```

```
variable "instance_name" {
  default = ["dev-server", "prod-server", "qa-server"]
}
```

```
variable "instance_type" {
  default = "t2.micro"
```

```
}
```

for

for loop

The for loop is designed for selecting elements from complex collections and performing operations on them. Assume you have a list of words (strings), and you want to convert them all into capitals in on go.

```
variable "country_list" {  
  default = ["Amercia","JaPan","InDia","LONDON"]  
}  
  
output "upper_country_list" {  
  value = [for country in var.country_list : upper(country)]  
}
```

To convert them all into capitals And it would result in:

```
["AMERICA", "JAPAN", "INDIA", "LONDON"]
```

The for expression's output is dependent on the type of input and the brackets around it. It would have produced a map if you had enclosed it in curly brackets and used a map as the input.

In above example you have seen that we have given input as list and we got the output as list only.

Handling maps using for loop

Let's see what happens if we give map as input and produce its output as list or give list as input and produce output as map using for loop

Map to List

Example:

```
locals {  
  list = {a = 1, b = 2, c = 3}  
}  
output "result1" {  
  value = [for k,v in local.list : "${k}-${v}" ]  
}  
output "result2" {  
  value = [for k in local.list : k ]  
}
```

Results:

```
result1 = ["a-1", "b-2", "c-3"]
```

```
result2 = [1, 2, 3]
```

In the above example, k denotes the key and v denotes the value. So, when the input source is a Map, you can access the key and value. We got output as list because of [] square brackets.

List to Map

To return the map type output, the output value should be enclosed in {} curly brackets.

Example:

```
locals {  
  list = ["a","b","c"]  
}  
output "result" {  
  value = {for i in local.list : i => i }  
}
```

Result:

```
result = {  
  "a" = "a"  
  "b" = "b"  
  "c" = "c"  
}
```

As you can see, a Map is now the result output type. It's crucial to keep in mind that when returning a Map, we must include an expression with the ==>. You'll experience the following error if you don't:

```
Error: Invalid 'for' expression  
on main.tf line 5, in output "result":  
5: value = {for i in local.list : i }
```

Map to Map

For Map to map result, the input and output both should be enclosed in curly {} brackets.

Example:

```
locals {  
  list = {a = 1, b = 2, c = 3}  
}  
output "result" {  
  value = {for k,v in local.list : k ==> v }  
}
```

Result:

```
result = {
  "a" = 1
  "b" = 2
  "c" = 3
}
```

So, when a for expression is wrapped around [] square brackets, then the result would be in list, when a for expression is wrapped around curly brackets {}, then the result would be in map and most importantly, when you are returning output as map, the expression elements should be separated by =>.

Handling filters using for loop

One more thing which is also very interesting about for loops is that you can use the for expression to filter the input anyway you like. You can conditionally operate on particular values or not, depending on your stated terms, by including an if clause.

Filter Simple Elements

You want to remove newline character from every element except “London.”

```
country_list = [
  "Amercia\n",
  "jaPan\n",
  "InDia\n",
  "London"
]
```

```
[for country in var.country_list : chomp(country) if country != "London"]
```

Filter Map Elements

In this example, the elements are Maps. We'll transform the data to only return the values, but only if the b key's value is greater than 6.

```
locals {
  list = [
    {a = 1, b = 5},
    {a = 2, b = 6},
    {a = 3, b = 7},
    {a = 4, b = 8},
  ]
}
output "list" {
  value = [for m in local.list : values(m) if m.b > 6 ]
}
```

Result:

```
list = [  
  [3,7]  
  [4,8],  
]
```

Filter Inconsistent Map Elements

Let's say the Map elements are a little bit inconsistent, like some of them lack the b key.
We can filter for that.

```
locals {  
  list = [  
    {a = 1, b = 5},  
    {a = 2},  
    {a = 3},  
    {a = 4, b = 8},  
  ]  
}  
output "list" {  
  value = [for m in local.list : m if contains(keys(m), "b")] ]  
}
```

Result:

```
list = [  
  {"a" = 1, "b" = 5},  
  {"a" = 4, "b" = 8},  
]
```


Dynamic block
Terraform taint
terragrunt