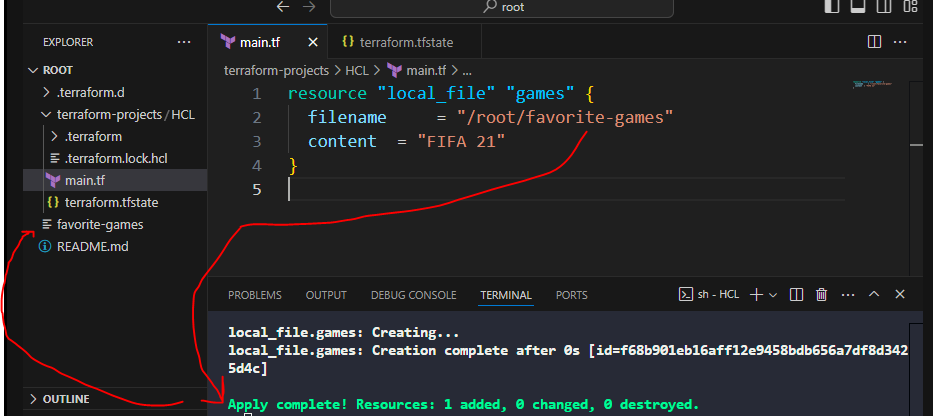
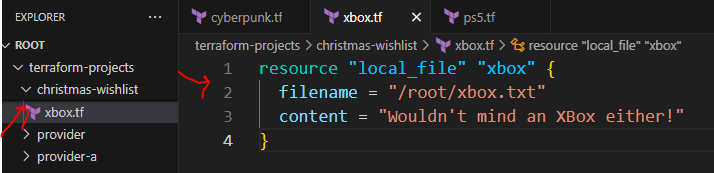
1. **A sample code:- where we create a file using provider local after successfully running terraform apply**



1. **Understand different component of terraform configuration file**

Create a new configuration file within the same directory called xbox.tf. This file should make use of the same **local\_file resource type** with the below requirements:

**Resource Name: xbox**  
  
**filename:** /root/xbox.txt  
  
**content:** Wouldn't mind an XBox either!  
  
  
Once the configuration file has been created, use the terraform workflow to create this resource.

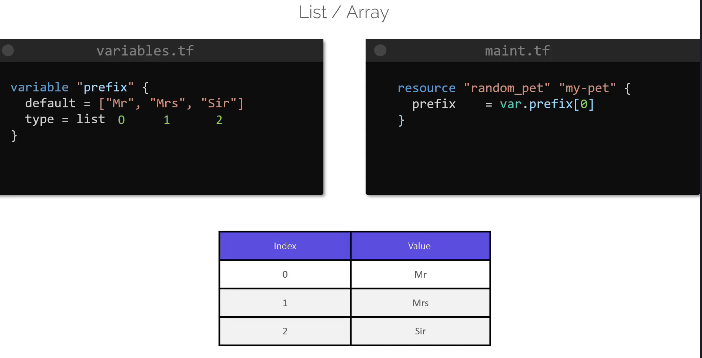


1. **Multiple providers:-**

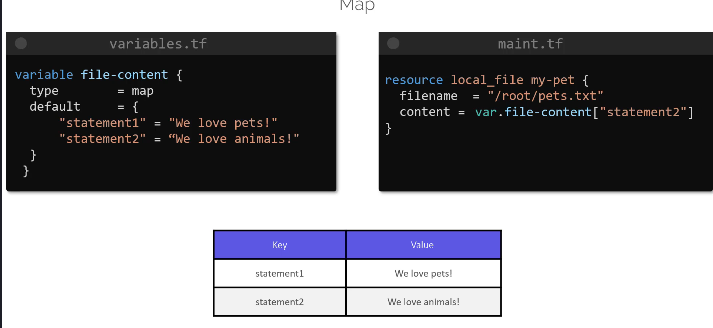
When we do terraform init on this code, we can see 2 providers plugin got created under .terraform/providers

1. Different type of variables and how to declare those and use in main.tf

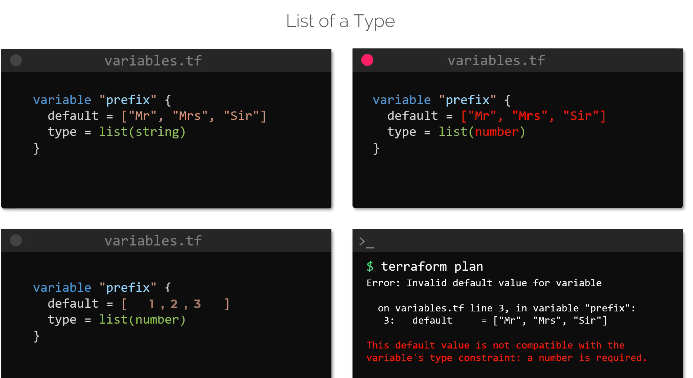
4.1 list



4.2 Map



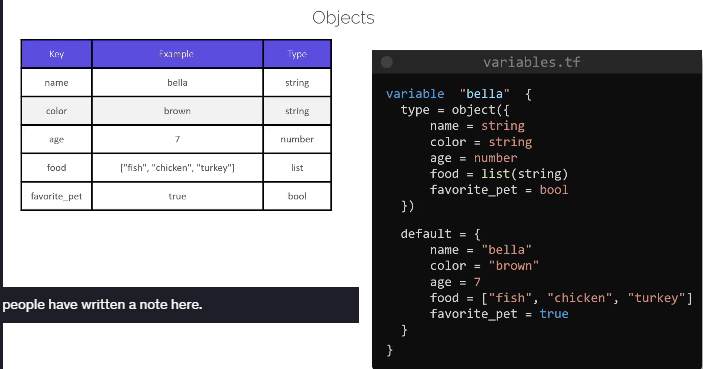
4.3 list of a type



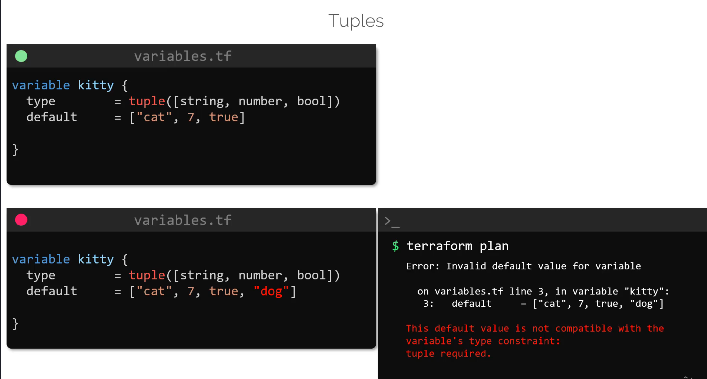
4.4 set is similar to list. But it cannot contain duplicate values.



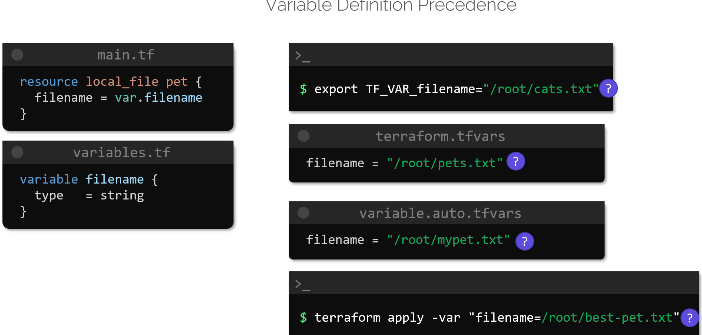
4.5 object

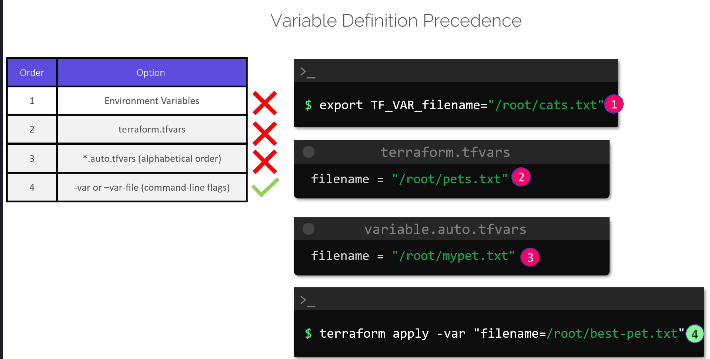


4.6 Tuple

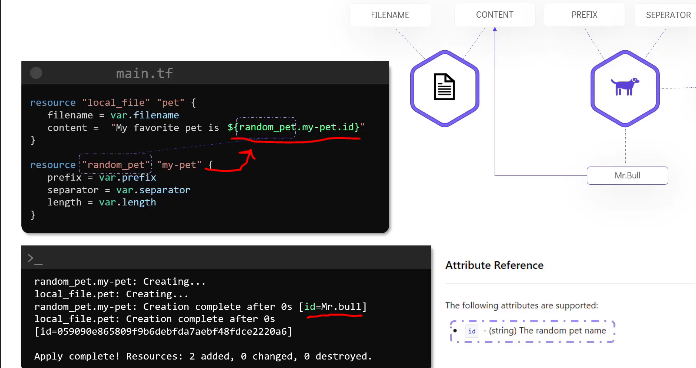


1. Variable definition Precedence:- if i am not mentioning the variable in variables.tf , these are the options i can define the variable. If the variable is defined multiple places then who would terraform give priority to.





1. Resource Attribute:-



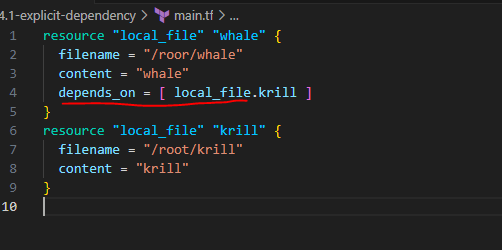
Here we are using the output of random\_pet as an input in local\_file

1. Resource dependency:-

Within this directory, create two local\_file type resources in main.tf file.  
  
  
Resource 1:  
  
Resource Name: whale  
  
File Name: /root/whale  
  
content: whale  
  
  
Resource 2:  
  
Resource Name: krill  
  
File Name: /root/krill

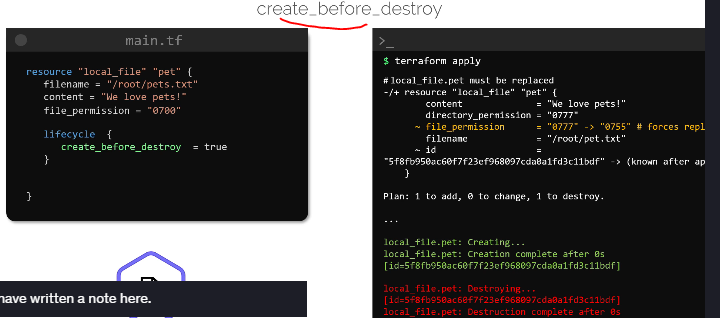
content: krill  
  
  
Resource called whale should depend on krill but do not use reference expressions.

When ready, run terraform init, plan and apply

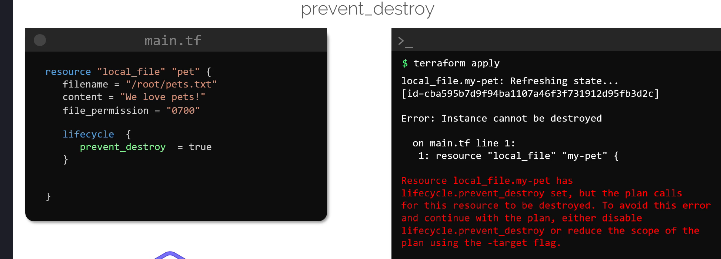


1. LifeCycle Rules:-

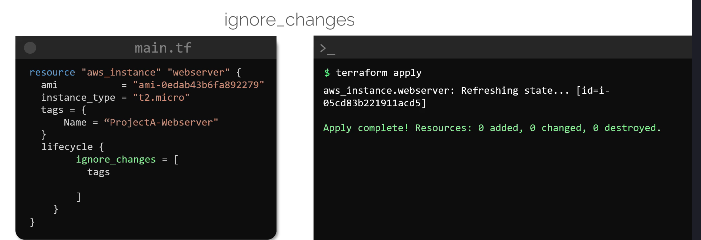
8.1 create before destroy



8.2 prevent destroy

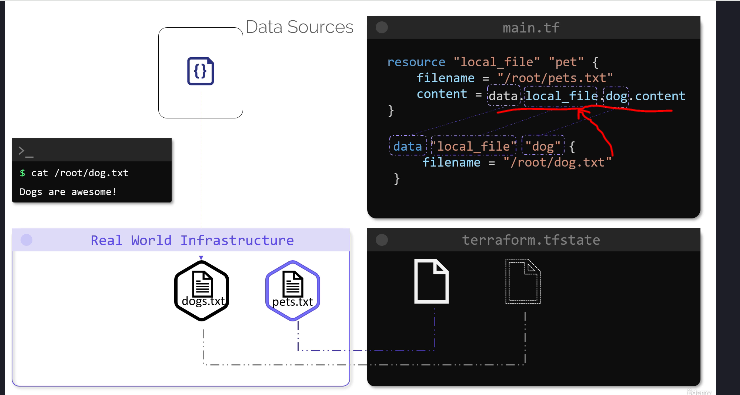


8.3 ignore\_changes

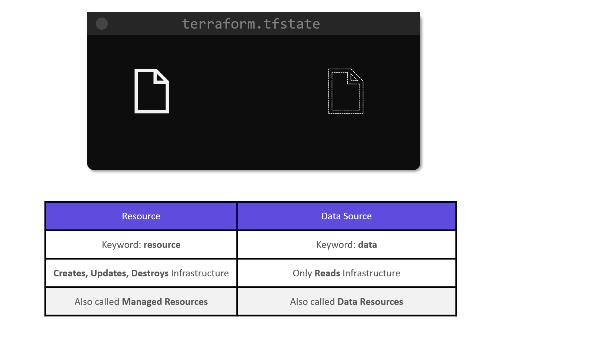


1. Data Sources:-

In the below example we created a local\_file resource named pet using terraform under root directory. But then we manually created another local\_file resource using bash named dog under root directory. Now terraform has no idea about this resource, since it is created manually. Now if terraform wants to use it, the resource 1st need to be declared as a data block in terraform then we can use that resource attributes in other terraform resources.

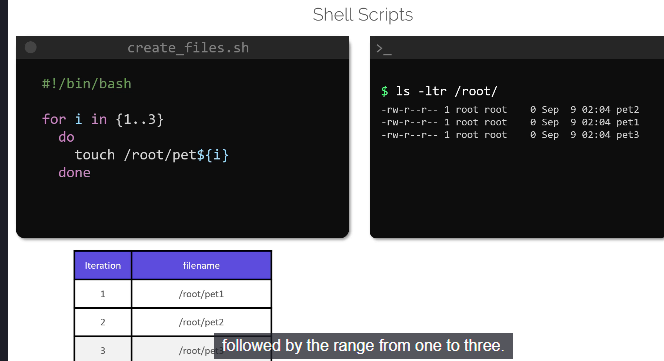


Difference between a resource and Data source

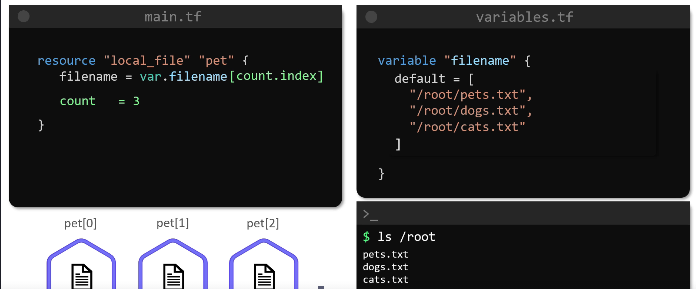


1. Meta-arguments. Count, for-each

When we want to create multiple instances of a resource we use count, for-each. Below is a shell script how we can create multiple resources using a for loop

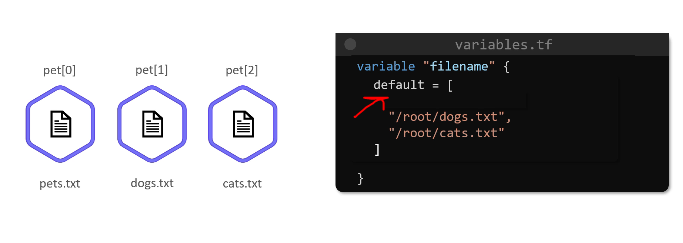


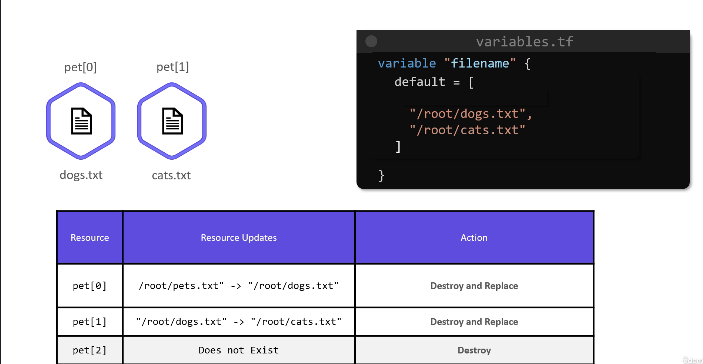
How we can replicate the above scenario using count .



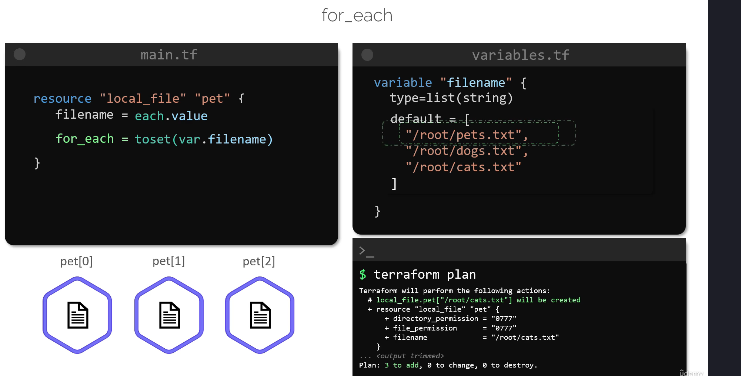


Disadvantage of count:- Since count is creating a list of resources, if i delete one resource from the code, terraform apply is not working accordingly.

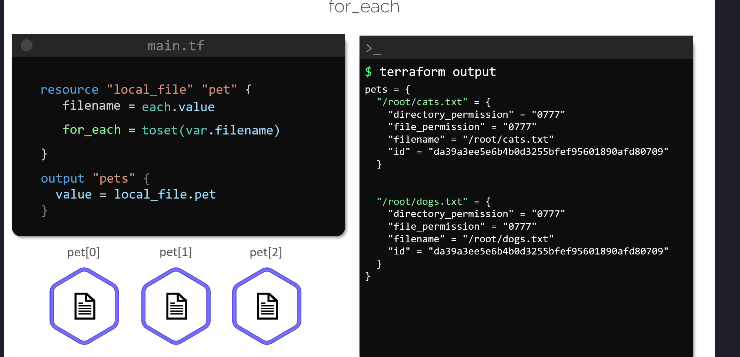


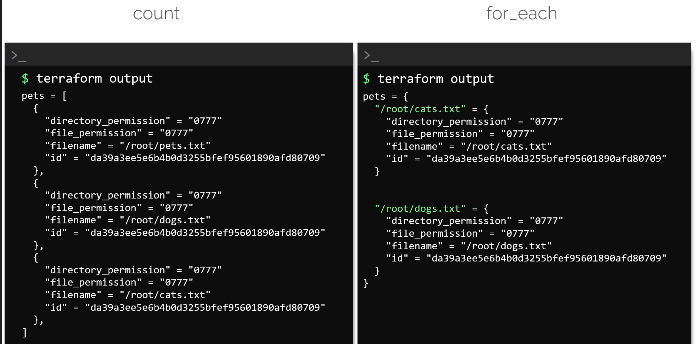


For\_each

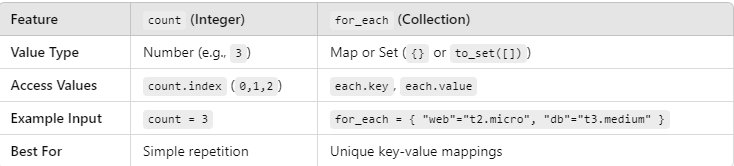


Now resources are treated as a key. for\_each would create a set of resources.

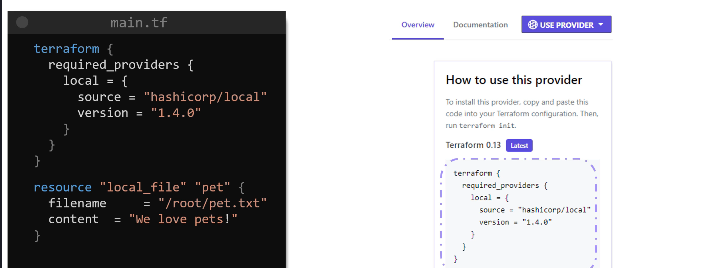




In **Terraform**, both count and for\_each are used to **create multiple resources dynamically**, but they have different use cases and behaviors.

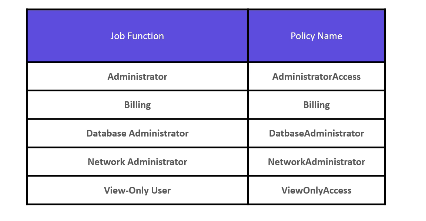


1. Provider block



AWS with Terraform:-

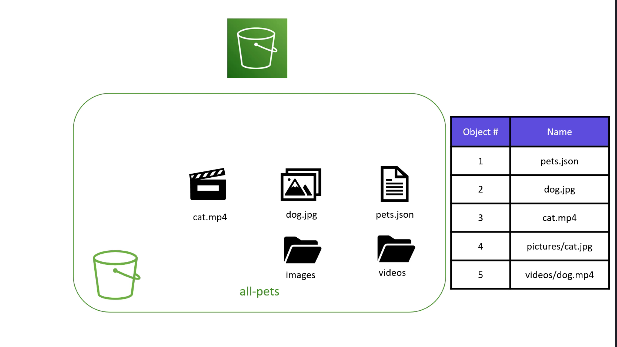
1. Different aws managed policies:-



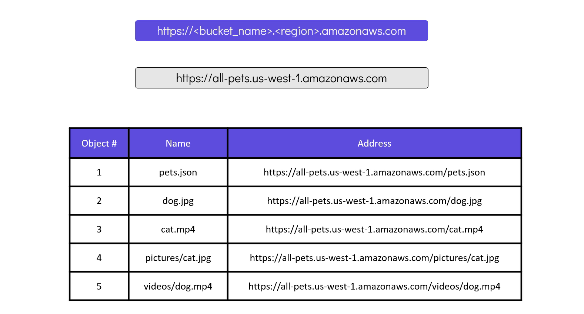
1. Project1 - create iam user, iam policy, attach policy to the user.



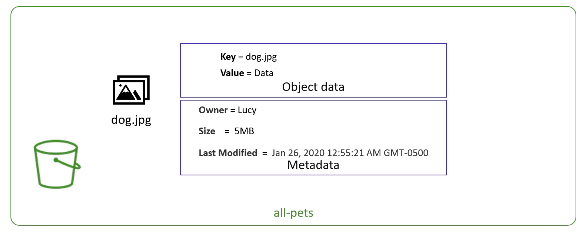
1. S3 bucket => in s3 data stored inside a bucket. Whatever there in a bucket, is object.



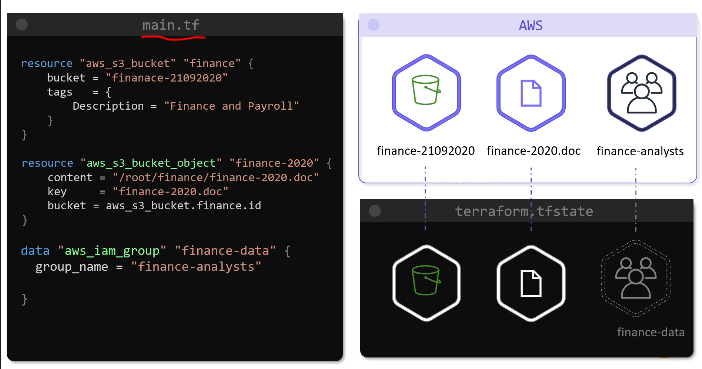
How we can access an s3 bucket and its objcts through its dns name

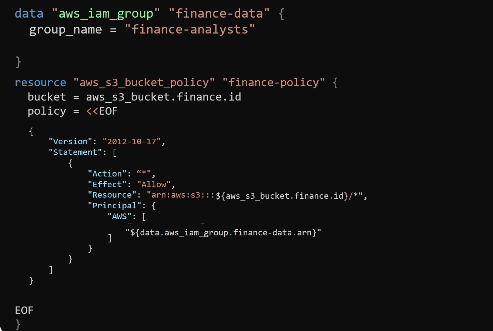


Any object in aws s3 bucket consist of object data and metadata. The data consist of key i.e actual name of the object and value which is actual data referenced by that object. Metadata consist of – the time when object was created, the owner, the size of the object.



1. Project2 – create s3 bucket => add object to the bucket => import an iam group as data source => attach bucket policy to the group so that it can acccess the bucket object



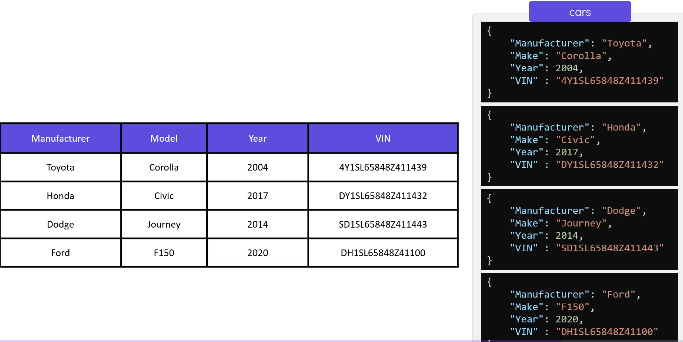


1. DynamoDB



The data here is stored as a table, which is a collection of key-value pairs. There are 4 keys now in the table. Each entry in this database is called an item, and in this example, we have a database of cars with four items at this moment.

Each item in this database can also be represented like this.



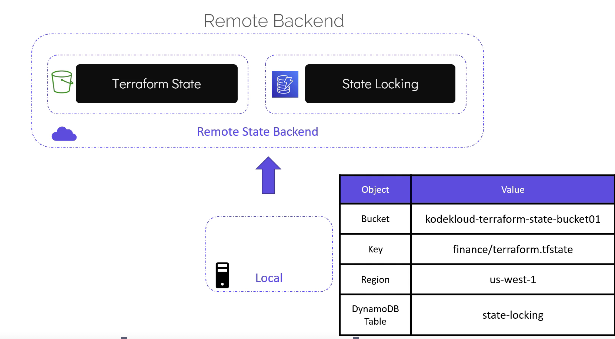
In order to uniquely distinguish an item with every other item in the table, DynamoDB uses a primary key. For example, the attribute called VIN or the vehicle identification number is a unique value, and this can be used to identify any car in this table.

A VIN in this case can be used as a primary key.

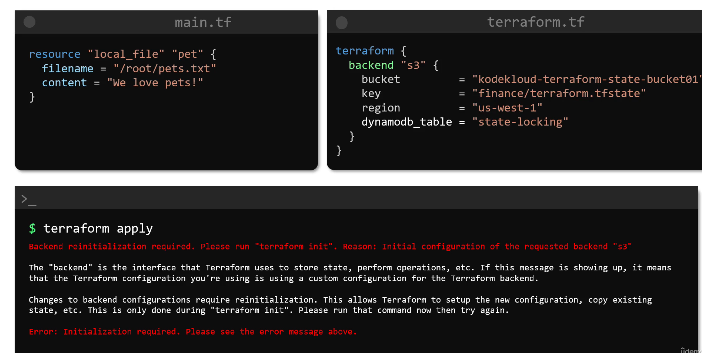
Project:- Create a DynamoDB table and add items to the table



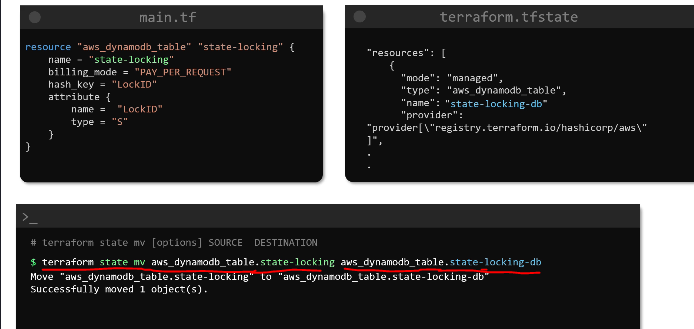
1. Configure Remote Backend – create the bucket and the table, then record these infos to use it in terraform code

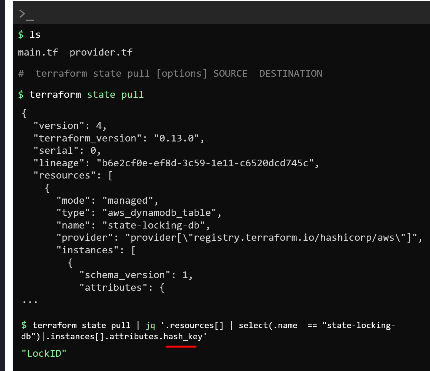




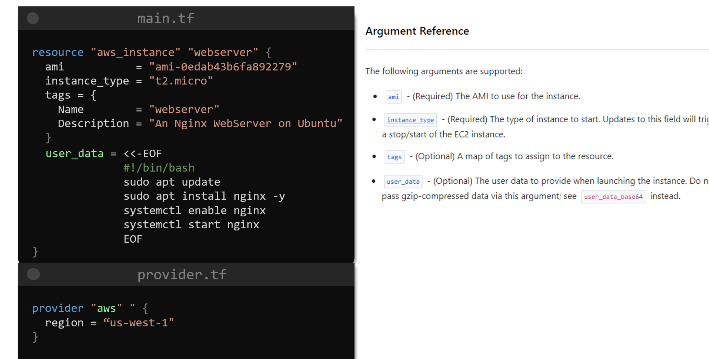


1. Terraform state commands

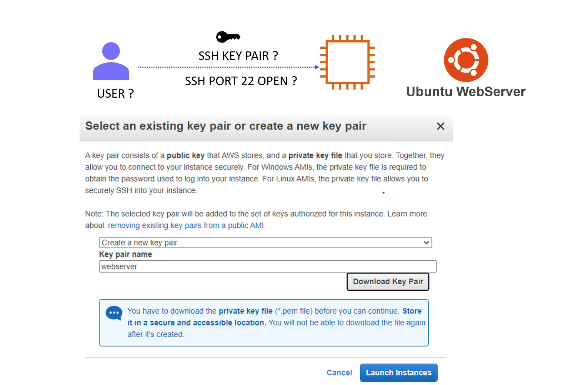




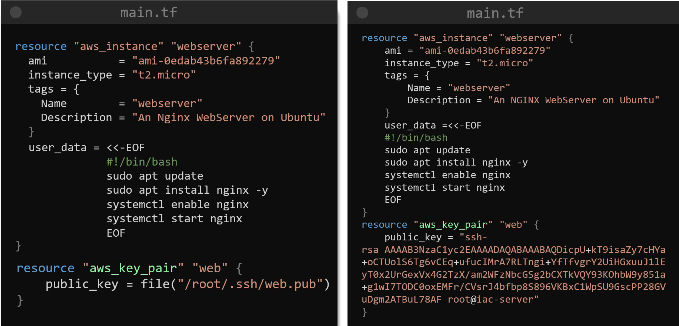
1. Create an ec2 instance using terraform



To ssh into it we need to provide key details



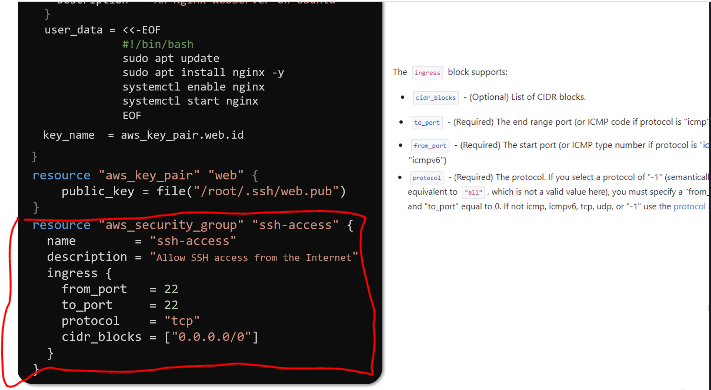
Now the modified code would be:-



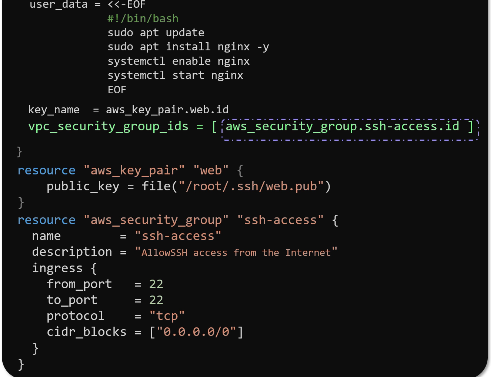
Then we need to provide the key details in user data to provide **key based access control** for the ec2 instance



Add resource for security group so that ssh at port 22 should be allowed from internet

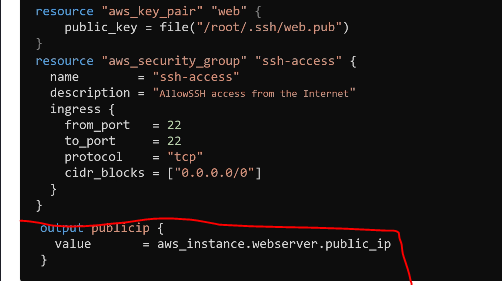


Now we need add this in the resource block of the ec2 instance

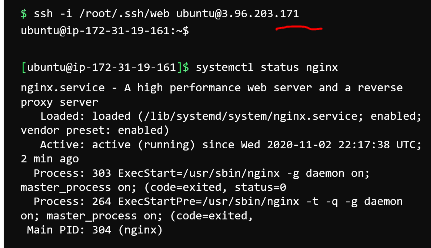


use a reference expression to specify the ID of the security group that we just configured. This argument expects a list. Make sure that the values are supplied within square brackets like this.

Get the public ip of the web server, we can use it later to ssh into the web server from our local machine

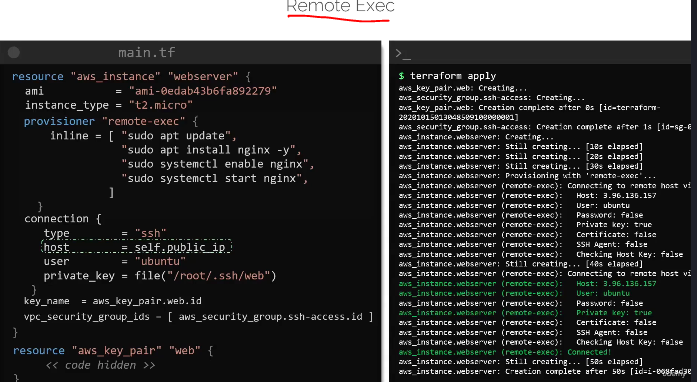


Once the servers is created, use the private key to ssh into it

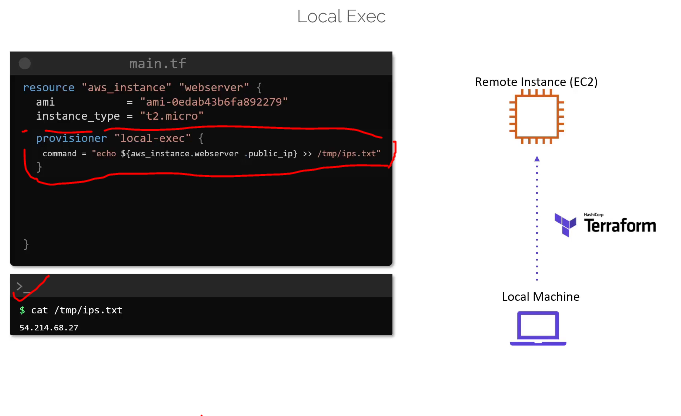


Provisioner;-

9. 1 Remote-exec provisioner



9.2 provisioner local-exec



9.3 considerations for using provisioners

* **Use Provisioners Sparingly**

Terraform advises minimal use of provisioners due to increased complexity.

Terraform cannot accurately model provisioner actions in the terraform plan.

* **Challenges with Provisioners**

Provisioners can execute any system-supported command, making execution unpredictable.

Some provisioners (e.g., remote-exec) require a **connection block**, which needs network connectivity and authentication.

* **Preferred Alternative: Native Resource Features**

Use built-in resource features instead of provisioners (e.g., **User Data for AWS EC2**).

These features run during resource creation without requiring a connection block.

* **Best Practice: Use Preconfigured Images**

Minimize post-provisioning tasks by using **custom images** with required software pre-installed.

Example: Instead of using remote-exec to install **NGINX**, use a custom AMI with NGINX pre-installed.

**2. Terraform Import project:-**

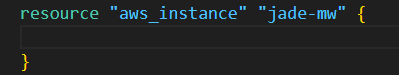
* Let's manage this instance called jade-mw with Terraform! First, create an empty resource block for this instance in the main.tf file in the configuration directory /root/terraform-projects/project-jade . Use jade-mw as the resource name. We will fill in the arguments for this block later.

Now to run import command we need the instance id, so i would be using aws cli to find instance id first using below command

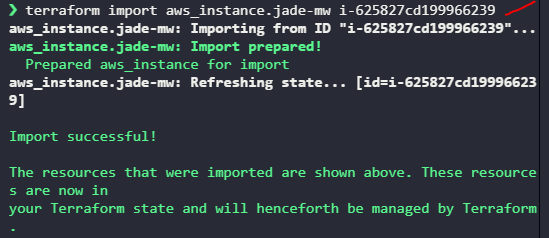
aws ec2 describe-instances --endpoint http://aws:4566 --filters "Name=image-id,Values=ami-082b3eca746b12a89" | jq -r '.Reservations[].Instances[].InstanceId'

[ we need to know the ami id] => output would be - i-625827cd199966239

Then create an empty resource block in main.tf

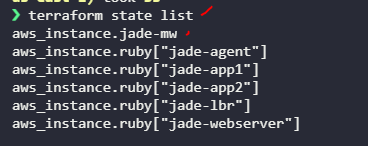


Then run the terraform import command



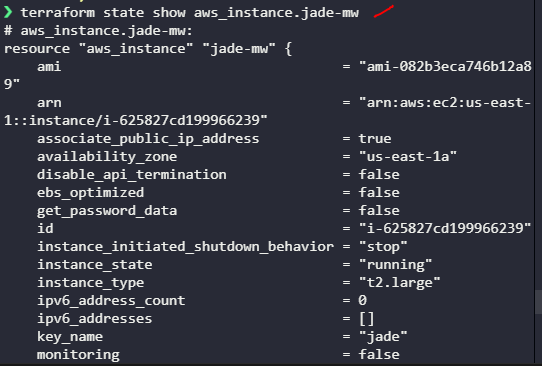
Once import successful the resource is available in terraform.tfstate

Terraform state list => would ensure that

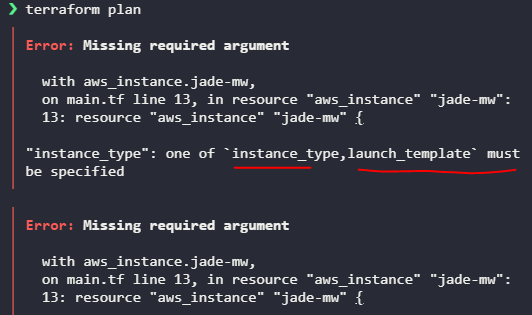


If u want to know all the arguments and its values of the resource then run,

Terraform state show aws\_instance.jade.mw

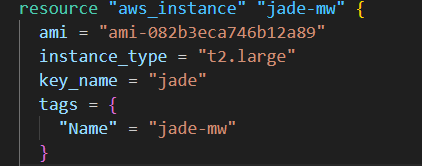


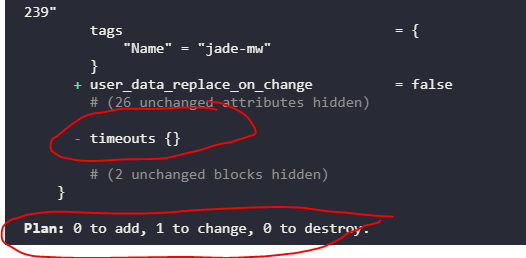
Now run terraform plan – it would ask for mandatory arguments



Now mention the mandatory arguments in resource block and run terraform apply

So mentioned these details and run terraform plan

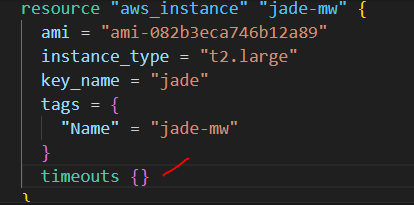




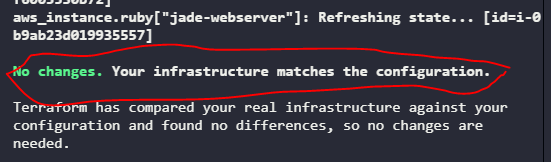
See terraform plan showing it would update the resource, since timeouts block not there.

But thats not intended. We want terraform plan to mention no changes required, so now checking the resource block from terraform state we would add the timeouts block

Now the resource looks like this



And terraform plan shows no changes required



Now we can run terraform apply => it would not throw any error

# **Terraform Modules: A Comprehensive Guide**

## **1. Introduction to Terraform Modules**

Terraform modules help in organizing and reusing infrastructure as code. They enable users to manage complex configurations efficiently by encapsulating resources into modular components.

## **2. Benefits of Using Terraform Modules**

✅ **Reusability** – Define infrastructure once and reuse it across multiple environments.

✅ **Maintainability** – Keep configurations structured and manageable.

✅ **Consistency** – Standardize infrastructure deployment across teams.

✅ **Scalability** – Deploy similar stacks in different regions or accounts easily.

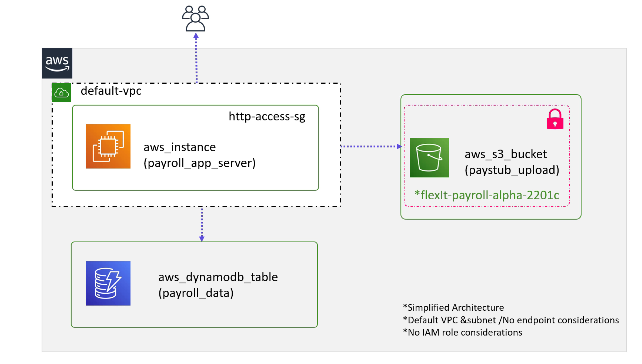
✅ **Collaboration** – Share modules among teams or use publicly available modules from the Terraform Registry.

## **3. Creating a Local Terraform Module**

### **Scenario: Deploying a Payroll Application**

An organization, **Flex IT Consulting**, wants to deploy a **payroll application** in multiple countries. The infrastructure consists of:

* **EC2 Instance** (Application Server) using a custom AMI.
* **DynamoDB Table** (NoSQL Database) to store employee and payroll data.
* **S3 Bucket** to store payroll documents (e.g., pay stubs, tax forms).



### **Module Structure**

The module is created under the modules/ directory:

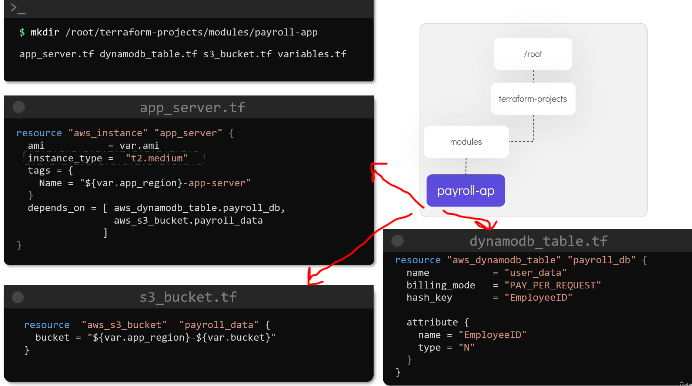
/root/terraform-project/  
├── modules/  
│ ├── payroll-app/  
│ │ ├── main.tf  
│ │ ├── variables.tf  
│ │ ├── outputs.tf  
│ │ ├── providers.tf  
│ ├── other-modules/  
├── us-payroll-app/  
│ ├── main.tf  
├── uk-payroll-app/  
│ ├── main.tf

### **Example: main.tf (Payroll App Module)**

resource "aws\_instance" "app\_server" {  
 ami = var.ami  
 instance\_type = "t3.micro"  
 tags = {  
 Name = "${var.app\_region}-payroll-app"  
 }  
 depends\_on = [aws\_s3\_bucket.payroll\_bucket, aws\_dynamodb\_table.payroll\_db]  
}  
  
resource "aws\_s3\_bucket" "payroll\_bucket" {  
 bucket = "${var.app\_region}-payroll-docs"  
}  
  
resource "aws\_dynamodb\_table" "payroll\_db" {  
 name = "payroll\_table"  
 hash\_key = "employee\_id"  
 billing\_mode = "PAY\_PER\_REQUEST"  
 attribute {  
 name = "employee\_id"  
 type = "S"  
 }  
}

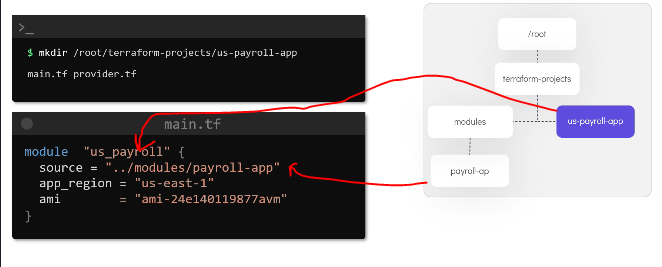
### **Example: variables.tf**

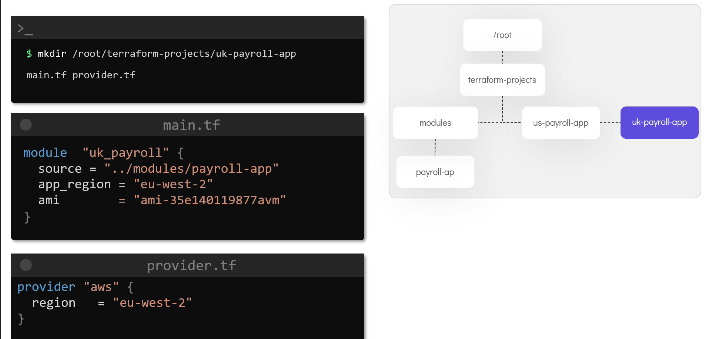
variable "app\_region" {  
 type = string  
}  
  
variable "ami" {  
 type = string  
}  
  
variable "bucket" {  
 type = string  
 default = "flexit-payroll-alpha-22001c"  
}



### **Using the Module in the US Region**

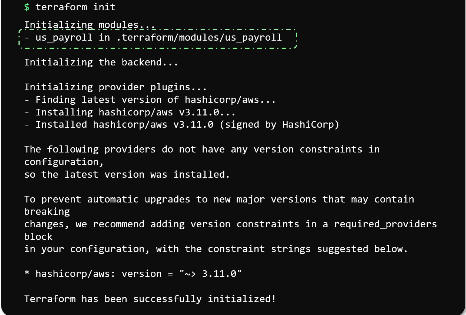
module "us\_payroll" {  
 source = "../modules/payroll-app"  
 app\_region = "us-east-1"  
 ami = "ami-0abcdef1234567890"  
}





### **Deploying the Module**

terraform init # Initialize Terraform and download module dependencies



terraform plan # Preview infrastructure changes  
terraform apply # Deploy the infrastructure

This module can be reused to deploy the same infrastructure in the UK by changing the app\_region and ami values.

## **4. Using Terraform Modules from the Terraform Registry**

### **What is the Terraform Registry?**

* A **public repository** for Terraform **providers** and **modules**.
* Modules are categorized based on the **cloud provider** (AWS, Azure, GCP, etc.).
* Two types of modules:
  + ✅ **Verified Modules** – Maintained & tested by HashiCorp.
  + 🏗 **Community Modules** – Published by users (not verified by HashiCorp).

### **Example: Using a Public Security Group Module**

To create a **security group** that allows **SSH access**, use the AWS security group module:

module "ssh\_security\_group" {  
 source = "terraform-aws-modules/security-group/aws"  
 version = "4.0.0"  
  
 name = "ssh-sg"  
 vpc\_id = "vpc-123456"  
 ingress\_cidr\_blocks = ["0.0.0.0/0"]  
 ingress\_rules = ["ssh-tcp"]  
}

### **Downloading and Applying the Module**

terraform init # Downloads the module from the Terraform Registry  
terraform plan # Preview changes  
terraform apply # Deploy security group

By specifying the **version**, we ensure stability in our infrastructure.

## **5. Benefits of Using Terraform Modules from the Registry**

✅ **Saves Time** – No need to build everything from scratch.

✅ **Standardization** – Uses **tested** and **validated** modules.

✅ **Version Control** – Ensures consistent deployments.

✅ **Scalability** – Easily deploys infrastructure across different environments.

## **6. Conclusion**

Terraform modules are a powerful way to organize, reuse, and maintain infrastructure as code. Whether using **local modules** for custom implementations or **public modules** from the Terraform Registry, they enhance efficiency, consistency, and scalability. Start building and utilizing modules to simplify your Terraform workflow!

# **Terraform Functions - Detailed Notes**

## **1️⃣ Overview of Terraform Functions**

Terraform provides **built-in functions** to manipulate and transform data within configuration files. These functions help in working with **strings, numbers, collections (lists, sets, maps), and type conversions**.

✅ **Commonly Used Terraform Functions**:

* **File Function** (file()) – Reads data from a file.
* **Length Function** (length()) – Counts elements in a list or map.
* **ToSet Function** (toset()) – Converts a list to a set (removes duplicates).

### **Testing Terraform Functions with Terraform Console**

Terraform provides an **interactive console** to test functions and interpolations before using them in configurations.

bash

CopyEdit

terraform console

This console loads **the Terraform state and variables** from the configuration directory, allowing real-time experimentation.

## **2️⃣ Terraform Numeric Functions**

Numeric functions help manipulate **numbers** in Terraform.

### **📌 Common Numeric Functions**

|  |  |  |  |
| --- | --- | --- | --- |
| **Function** | **Description** | **Example** | **Result** |
| max() | Returns the largest number | max(10, 20, 30) | 30 |
| min() | Returns the smallest number | min(10, 20, 30) | 10 |
| ceil() | Rounds up to the nearest whole number | ceil(10.1) | 11 |
| floor() | Rounds down to the nearest whole number | floor(10.9) | 10 |

### **🔹 Expanding Variables in Functions**

Terraform allows passing variables into functions using an **expansion symbol (...)**.  
Example:

hcl

CopyEdit

variable "numbers" {  
 default = [5, 10, 15]  
}  
output "max\_value" {  
 value = max(var.numbers...)  
}

## **3️⃣ Terraform String Functions**

String functions allow **modifying and manipulating strings**.

### **📌 Common String Functions**

|  |  |  |  |
| --- | --- | --- | --- |
| **Function** | **Description** | **Example** | **Result** |
| split() | Splits a string into a list using a delimiter | split(",", "ami-123,ami-456,ami-789") | ["ami-123", "ami-456", "ami-789"] |
| join() | Joins a list into a string | join("-", ["hello", "world"]) | "hello-world" |
| lower() | Converts a string to lowercase | lower("HELLO") | "hello" |
| upper() | Converts a string to uppercase | upper("hello") | "HELLO" |
| title() | Capitalizes the first letter of each word | title("hello world") | "Hello World" |
| substr() | Extracts part of a string using offset and length | substr("abcdef", 2, 3) | "cde" |

### **🔹 Example: Splitting and Joining Strings**

hcl

CopyEdit

variable "ami\_list" {  
 default = "ami-123,ami-456,ami-789"  
}  
  
output "split\_amis" {  
 value = split(",", var.ami\_list)  
}  
  
output "joined\_amis" {  
 value = join("-", split(",", var.ami\_list))  
}

## **4️⃣ Terraform Collection Functions (Lists, Sets, Maps)**

Collection functions help **manipulate lists, sets, and maps** in Terraform.

### **📌 List Functions**

|  |  |  |  |
| --- | --- | --- | --- |
| **Function** | **Description** | **Example** | **Result** |
| length() | Gets the number of elements in a list | length(["a", "b", "c"]) | 3 |
| toset() | Converts a list to a set (removes duplicates) | toset(["us-east-1", "us-west-1", "us-east-1"]) | ["us-east-1", "us-west-1"] |
| index() | Gets the index of an item in a list | index(["ami-123", "ami-456"], "ami-456") | 1 |
| element() | Gets an item at a given index | element(["ami-123", "ami-456"], 1) | "ami-456" |
| contains() | Checks if a value exists in a list | contains(["ami-123", "ami-456"], "ami-456") | true |

### **🔹 Example: Using length(), toset(), and contains()**

hcl

CopyEdit

variable "regions" {  
 default = ["us-east-1", "us-west-1", "us-east-1"]  
}  
  
output "unique\_regions" {  
 value = toset(var.regions) # Removes duplicates  
}  
  
output "num\_regions" {  
 value = length(var.regions) # Returns 3  
}  
  
output "is\_present" {  
 value = contains(var.regions, "us-west-1") # Returns true  
}

### **📌 Map Functions**

|  |  |  |  |
| --- | --- | --- | --- |
| **Function** | **Description** | **Example** | **Result** |
| keys() | Extracts keys from a map | keys({ "us-east-1" = "ami-123", "us-west-1" = "ami-456" }) | ["us-east-1", "us-west-1"] |
| values() | Extracts values from a map | values({ "us-east-1" = "ami-123", "us-west-1" = "ami-456" }) | ["ami-123", "ami-456"] |
| lookup() | Gets a value for a key | lookup({ "us-east-1" = "ami-123" }, "us-east-1") | "ami-123" |

### **🔹 Example: Working with Maps**

hcl

CopyEdit

variable "ami\_map" {  
 default = {  
 "us-east-1" = "ami-123"  
 "us-west-1" = "ami-456"  
 }  
}  
  
output "all\_keys" {  
 value = keys(var.ami\_map) # Returns ["us-east-1", "us-west-1"]  
}  
  
output "all\_values" {  
 value = values(var.ami\_map) # Returns ["ami-123", "ami-456"]  
}  
  
output "specific\_ami" {  
 value = lookup(var.ami\_map, "us-east-1") # Returns "ami-123"  
}

✅ **Using a Default Value in lookup()**:

hcl

CopyEdit

output "safe\_lookup" {  
 value = lookup(var.ami\_map, "us-west-2", "ami-default") # Returns "ami-default"  
}

## **5️⃣ Terraform Console Testing**

Terraform provides an **interactive console** to test functions before using them in configurations.

### **📌 Start the Console**

bash

CopyEdit

terraform console

### **📌 Test Functions**

bash

CopyEdit

> length(["a", "b", "c"])  
3  
  
> split(",", "ami-123,ami-456,ami-789")  
["ami-123", "ami-456", "ami-789"]  
  
> toset(["us-east-1", "us-west-1", "us-east-1"])  
toset(["us-east-1", "us-west-1"])  
  
> lookup({ "us-east-1" = "ami-123" }, "us-east-1")  
"ami-123"

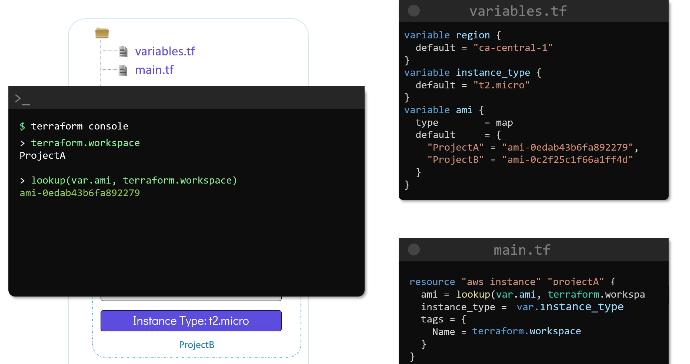
# **🔟 Summary of Key Terraform Functions**

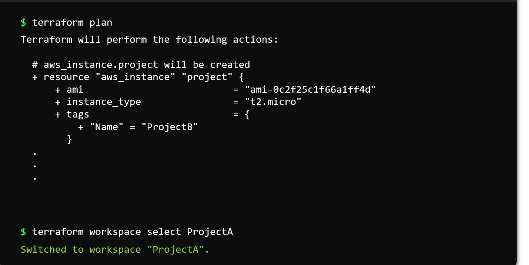
|  |  |
| --- | --- |
| **Category** | **Functions** |
| **Numeric** | max(), min(), ceil(), floor() |
| **String** | split(), join(), lower(), upper(), title(), substr() |
| **List** | length(), toset(), index(), element(), contains() |
| **Map** | keys(), values(), lookup() |

## **✅ Best Practices**

* **Use Terraform Console** for testing before applying changes.
* **Avoid hardcoding values**; use **variables** where possible.
* **Understand function limitations** (e.g., toset() removes duplicates).
* **Use lookup() with a default value** to avoid errors.

**Terraform workspaces**





Project:-

Let's now write the main.tf file to make use of the same module that we saw in the terraform modules lecture.  
  
The project-sapphire configuration directory will be used to deploy the same payroll application stack in different regions.  
  
The module is located at the path /root/terraform-projects/modules/payroll-app

Now, update the main.tf of the root module to call the child module located at /root/terraform-projects/modules/payroll-app. Adhere to the following specifications:

1. module name: payroll\_app
2. This module expects two mandatory arguments:  
     
   a. app\_region - use the values from variable called region  
     
   b. ami - use the values from the variable called ami
3. The values for these two arguments should be selected based on the workspace you are on.  
     
     
   For example, if on us-payroll workspace, the app\_region should be us-east-1 and the ami ami-24e140119877avm OR for uk-payroll, the app\_region should be eu-west-2 and the ami ami-35e140119877avm e.t.c .

Once ready, run terraform init. You don't have to create(apply) the resources yet!  
Refer to the README.md file located at /root/terraform-projects/modules/payroll-app to see how to use this module.

Now, using the same configuration, create the resources on all three workspaces that you created earlier!

|  |  |
| --- | --- |
| Main.tf | Variables.tf |
| module "payroll\_app" {  source = "../modules/payroll-app"  app\_region = lookup(var.region,terraform.workspace)  ami = lookup(var.ami,terraform.workspace)  } | variable "region" {  type = map  default = {  "us-payroll" = "us-east-1"  "uk-payroll" = "eu-west-2"  "india-payroll" = "ap-south-1"  }  }  variable "ami" {  type = map  default = {  "us-payroll" = "ami-24e140119877avm"  "uk-payroll" = "ami-35e140119877avm"  "india-payroll" = "ami-55140119877avm"  }  } |

|  |  |
| --- | --- |
|  |  |

terraform workspace new us-payroll

terraform workspace new uk-payroll

terraform workspace new india-payroll

==========

terraform workspace select us-payroll

Terraform apply

=======

terraform workspace select uk-payroll

Terraform apply

======

terraform workspace select india-payroll

Terraform apply

======

jkklll