TERRAFORM

What is Terraform?

Terraform is an open-source infrastructure as code (IaC) tool developed by HashiCorp that allows you to define and provision infrastructure using a declarative configuration language. When working with AWS (Amazon Web Services), Terraform helps you automate the process of setting up and managing your cloud resources.

Here's a quick overview of how Terraform works with AWS:

1. **Configuration**: You write Terraform configurations using the HashiCorp Configuration Language (HCL). In these configurations, you define the desired state of your infrastructure, including resources like EC2 instances, S3 buckets, RDS databases, and more.

2. **Execution**: Terraform uses these configuration files to interact with the AWS API. It generates an execution plan that shows what actions will be taken to achieve the desired state described in your configuration.

3. **Provisioning**: After reviewing the execution plan, you apply it. Terraform provisions the resources as specified and maintains them according to your configuration. It handles creating, updating, and deleting resources as needed.

4. **State Management**: Terraform keeps track of the infrastructure's current state in a state file. This file is crucial for managing changes and ensuring that Terraform knows what resources it is managing.

5. **Modularity and Reusability**: Terraform supports modules, which are reusable configurations that can help you manage complex infrastructure setups efficiently. You can use Terraform's pre-made setups (called modules) to make common tasks easier and share your setup with others.

Overall, Terraform provides a way to manage and automate cloud infrastructure in a consistent and repeatable manner, which can greatly simplify operations and reduce errors.

Benefits of Terraform

Using Terraform provides several significant benefits, especially when managing cloud infrastructure:

1. **Infrastructure as Code (IaC)**: Terraform allows you to define your infrastructure using code. This means you can version control your infrastructure setup, track changes, and collaborate with others just like you would with application code.

2. ****Automation****: Terraform automates the process of setting up and managing cloud resources. This reduces the manual effort required and helps to avoid human errors.

3. ****Consistency****: By using code to define your infrastructure, you ensure that environments are set up in a consistent manner. This consistency helps prevent configuration drift where different environments (e.g., development, staging, production) end up being different. It ensures that your different environments (like testing and production) are set up the same way, avoiding mistakes.

4. ****Scalability****: Terraform can manage complex infrastructure setups with ease. You can define, manage, and scale resources across multiple regions and accounts.

5. ****Reusability****: Terraform supports modules, which are reusable pieces of code. You can create standard templates for common setups and reuse them across different projects or environments.

6. ****Change Management****: Terraform provides a clear execution plan before making any changes. This plan shows you exactly what changes will be made, which helps you understand the impact and reduces the risk of unintended modifications.

7. ****State Management****: Terraform keeps track of the current state of your infrastructure. This state management allows Terraform to understand what resources are currently in place and how they should be updated or removed.

8. ****Multi-Provider Support****: Terraform is not limited to AWS; it supports multiple cloud providers and services (like Azure, Google Cloud, and more). This makes it easier to manage multi-cloud environments or integrate different services.

9. ****Improved Collaboration****: Since Terraform configurations are text files, they can be shared and reviewed by team members. This improves collaboration and ensures that everyone is on the same page regarding infrastructure changes.

10. ****Rollback Capability****: If something goes wrong, Terraform can help roll back changes to the previous state, reducing downtime and errors.

Overall, Terraform helps streamline infrastructure management, making it more predictable, efficient, and scalable.

Difference between TERRAFORM and ANSIBLE

Terraform : Terraform sets up the infrastructure: You use Terraform to create the servers, databases, and networks you need

- **Purpose**: Terraform is used to **provision** and **manage infrastructure**. This means it helps you set up cloud resources like servers, databases, and networks.

- **How It Works**: You write a configuration file that describes your desired infrastructure setup. Terraform then creates or updates these resources to match your configuration.
- **State Management**: Terraform keeps track of the current state of your infrastructure to know what resources exist and how they should be managed.
- **Usage**: Best for creating and managing the infrastructure itself (like setting up EC2 instances or S3 buckets).

Ansible : Ansible configures the infrastructure: After the infrastructure is set up, you use Ansible to install software, configure settings, and get your systems ready to use.

- **Purpose**: Ansible is used to **configure** and **manage** software and applications on existing infrastructure. This means it helps you set up and manage the software and settings on your servers.
- **How It Works**: You write playbooks (scripts) that describe how to configure your servers and applications. Ansible then runs these scripts to apply the configurations.
- **State Management**: Ansible does not keep track of the state of infrastructure; it focuses on making sure software and settings are configured as specified in your playbooks.
- **Usage**: Best for configuring and maintaining software on servers (like installing a web server or updating application settings).

Example Scenario

1. **Terraform (Provisioning)**:
    - You use Terraform to **create a virtual machine (VM)** in a cloud environment (like AWS EC2, Google Cloud Compute Engine, etc.). This is similar to setting up a laptop with its hardware and basic OS.
2. **Ansible (Configuration)**:
    - After the VM is created, you use Ansible to **install and configure software** on it, such as web servers, databases, or application code. This is similar to installing applications and configuring settings on a laptop.

Terraform Commands

When provisioning infrastructure using Terraform, a typical workflow involves several key commands to initialize, plan, and apply the infrastructure changes. Here are the Terraform commands commonly used during the infrastructure provisioning process:

1. `terraform init :`Prepares your environment by downloading necessary plugins and setting up the backend for state management.

2. `terraform plan :`Creates a preview of what Terraform will do, showing which resources will be created, updated, or deleted without making any changes.

3. **`terraform apply :`** Actually creates or updates the infrastructure based on your configuration. This is the step that provisions your resources.

4. **`terraform output`** (optional) : Shows the output values like resource URLs or IP addresses that you defined in your configuration.

5. **`terraform destroy`** (optional) : Removes or tears down all the infrastructure that was created.

Simplified Workflow:

Initialize the environment:

```
terraform init
```

Preview the changes:

```
terraform plan
```

Apply the changes to create the infrastructure:

```
terraform apply
```

(Optional) View output values:

```
terraform output
```

(Optional) Destroy the infrastructure:

```
terraform destroy
```

These commands make up the core workflow when provisioning infrastructure with Terraform, allowing you to initialize, review, apply, and manage your infrastructure as code.

PROVIDERS in Terraform

A **provider** in Terraform is a tool that allows Terraform to manage and interact with different services. It's like a bridge between Terraform and the service you want to work with, such as AWS, Azure, or Google Cloud. Provider block help Terraform to understand where it has to create the resources (AWS, AZURE, GCP).

How Providers Work

1. **Connect to Services**:
   - Providers connect Terraform to various services. For example, the AWS provider connects Terraform to AWS.
2. **Manage Resources**:
   - Providers know how to create, update, or delete resources on the service. For example, the AWS provider can create EC2 instances, S3 buckets, etc.
3. **Configure the Provider**:
   - You tell Terraform which provider to use and give it necessary details, like which region to work in or credentials to use.

Basic Example

1. **Specify the Provider**:
   - Here's how you tell Terraform to use the AWS provider and set the AWS region:

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

2. **Define Resources**:
   - With the AWS provider, you can define resources like an EC2 instance:

```
resource "aws_instance" "example" {
  ami           = "ami-12345678"  # Amazon Machine Image ID
  instance_type = "t2.micro"      # Instance type
}
```

3. **Output Information**:

- You can display information about the resources you create:

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

Summary

- **Provider**: Connects Terraform to a service (e.g., AWS).
- **Configuration**: Set up the provider with details like region or credentials.
- **Resources**: Define what you want to create or manage.
- **Outputs**: Show information about your resources.

In essence, providers let Terraform manage various services by providing the necessary connection and instructions for working with them.

How Terraform will create resources in multi-region

Terraform can manage resources across multiple regions by configuring the `provider` blocks and specifying the regions where resources should be created.

1. Define Multiple Providers

You need to define multiple `provider` blocks in your Terraform configuration file, each specifying a different region. Here's an example for AWS:

```
provider "aws" {
  alias  = "us_east_1"
  region = "us-east-1"
}

provider "aws" {
  alias  = "us_west_1"
  region = "us-west-1"
}
```

In this example, two providers are defined: one for the `us-east-1` region and another for the `us-west-1` region. You can make use of alias keyword to implement multi region infrastructure setup in terraform.

2. Specify Providers in Resource Definitions

When defining resources, you need to specify which provider should be used. This is done using the `provider` argument within the resource block:

```
resource "aws_instance" "example_us_east" {
  provider = aws.us_east_1
  ami           = "ami-0c55b159cbfafe1f0"  # Replace with a valid AMI ID
  instance_type = "t2.micro"
  # Other configurations...
}

resource "aws_instance" "example_us_west" {
  provider = aws.us_west_1
  ami           = "ami-0c55b159cbfafe1f0"  # Replace with a valid AMI ID
  instance_type = "t2.micro"
  # Other configurations...
}
```

In this configuration:

- The `aws_instance.example_us_east` resource will be created in the `us-east-1` region.
- The `aws_instance.example_us_west` resource will be created in the `us-west-1` region.

3. Initialize and Apply

After configuring the providers and resources, initialize your Terraform workspace and apply the configuration:

```
terraform init
terraform apply
```

Terraform will create resources in the specified regions based on your configuration.

Summary

- **Define multiple providers**: Create separate `provider` blocks for each region you need.
- **Reference providers in resources**: Use the `provider` argument within resource blocks to specify which provider (and thus which region) to use.

How Terraform will create resources in multi-cloud

Terraform can manage resources across multiple cloud providers, also known as multi-cloud environments, by configuring multiple provider blocks within the same configuration.

1. Define Providers

You need to define a separate `provider` block for each cloud provider you want to use. Each provider block specifies the details necessary to interact with that particular cloud provider.

For example, to manage resources in both AWS and Azure, you would configure them as follows:

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

provider "azurerm" {
  subscription_id = "your-azure-subscription-id"
  client_id = "your-azure-client-id"
  client_secret = "your-azure-client-secret"
  tenant_id = "your-azure-tenant-id"
}
```

2. Create Resources for Each Cloud Provider

You need to specify which provider each resource should use by referring to the appropriate `provider` alias in your resource definitions. Here's an example where we create an EC2 instance in AWS and a virtual machine in Azure:

```
# AWS EC2 Instance
resource "aws_instance" "example" {
  provider = aws
  ami           = "ami-0c55b159cbfafe1f0"  # Replace with a valid AMI ID
  instance_type = "t2.micro"
  # Other configurations...
}

# Azure Virtual Machine
resource "azurerm_virtual_machine" "example" {
  provider                = azure
  name                    = "example-vm"
  resource_group_name     = "example-resources"
  location                = "East US"
  size                    = "Standard_DS1_v2"
  network_interface_ids   = [azurerm_network_interface.example.id]
  vm_size                 = "Standard_DS1_v2"
  # Other configurations...
}
```

3. Initialize and Apply Configuration

Once your configuration is set up, initialize Terraform and apply the configuration:

```
terraform init
terraform apply
```

VARIABLES in Terraform

In Terraform, variables allow you to parameterize your configuration, making it more flexible and reusable. You can define variables to hold values that can be used throughout your configuration.

Types : Input variable - Input variables allow you to customize your Terraform configurations without hardcoding values. They make your setup flexible and reusable.

Output variable - Output variables let you see or use information from your Terraform configuration after resources are created. They are useful for sharing data between different Terraform configurations or modules.It can be used if a user wants to get the value of some resources.

INPUT Variable : Input variables are defined using the `variable` block in your Terraform configuration files.

```
variable "region" {
  description = "The AWS region to deploy resources into"
  type        = string
  default     = "us-east-1"
}

variable "instance_type" {
  description = "Type of EC2 instance"
  type        = string
  default     = "t2.micro"
}
```

You refer to these variables in your resource definitions using the `var` keyword:

```
provider "aws" {
  region = var.region
}

resource "aws_instance" "example" {
  ami           = "ami-0c55b159cbfafe1f0"  # Replace with a valid AMI ID
  instance_type = var.instance_type
}
```

Set Variable Values: **In `terraform.tfvars` File**: In Terraform, .tfvars files (typically with a .tfvars extension) are used to set specific values for input variables defined in your Terraform configuration.

```
region = "us-west-2"
```

OUTPUT Variable : Output variables are defined using the `output` block.

```
output "instance_id" {
  description = "The ID of the created EC2 instance"
  value       = aws_instance.example.id
}

output "instance_public_ip" {
  description = "The public IP address of the EC2 instance"
  value       = aws_instance.example.public_ip
}
```

Conditional Expression in Terraform

Conditional expressions in Terraform allow you to make decisions and vary configuration values based on certain conditions. They are useful for customizing your infrastructure based on different conditions without duplicating code.

The syntax for a conditional expression in Terraform is:

condition ? true_val : false_val

- `condition` is an expression that evaluates to either `true` or `false`.
- `true_val` is the value that is returned if the condition is `true`.
- `false_val` is the value that is returned if the condition is `false`.

```
variable "environment" {
  description = "Environment type"
  type        = string
  default     = "development"
}

variable "production_subnet_cidr" {
  description = "CIDR block for production subnet"
  type        = string
  default     = "10.0.1.0/24"
}

variable "development_subnet_cidr" {
  description = "CIDR block for development subnet"
  type        = string
  default     = "10.0.2.0/24"
}

resource "aws_security_group" "example" {
  name        = "example-sg"
  description = "Example security group"

  ingress {
    from_port   = 22
    to_port     = 22
    protocol    = "tcp"
    cidr_blocks = var.environment == "production" ? [var.production_subnet_cidr] : [var.development_subnet_cidr]
  }
}
```

Here, in the resource block, we are creating a security group and allowing ssh port which is 22 port number with a condition that if the variable environment value is equal to “production”, allow the subnet cidr from variable production. Otherwise, allow the subnet cidr from the development variable.

MODULES in Terraform

In Terraform, a **module** is a container for multiple resources that are used together. Modules help you organize and reuse code, making it easier to manage complex infrastructure and apply consistent patterns across different parts of your configuration.

Benefits of Using Modules

- **Reusability**: Write a module once and reuse it across different configurations and projects.
- **Organization**: Break down complex configurations into manageable parts.
- **Consistency**: Apply standard patterns and practices across different environments and teams.
- **Encapsulation**: Hide the complexity of resource setup behind a simple interface with input variables and outputs.

Structure: Typically includes `main.tf`, `variables.tf`, and `outputs.tf` files.

main.tf file

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

resource "aws_instance" "example" {
    ami = var.ami_value
    instance_type = var.instance_type_value
    subnet_id = var.subnet_id_value
}
```

variables.tf file

```
variable "ami_value" {
    description = "value for the ami"
}

variable "instance_type_value" {
    description = "value for instance_type"
}

variable "subnet_id_value" {
    description = "value for the subnet_id"
}
```

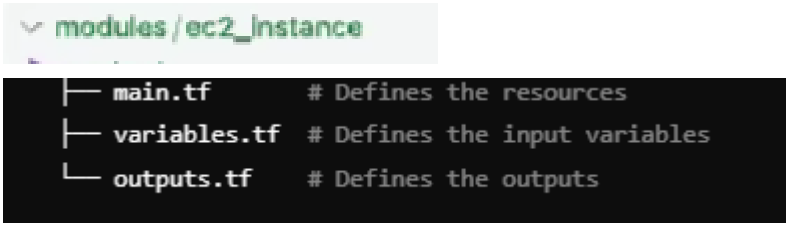
outputs.tf file

```
output "public-ip-address" {
  value = aws_instance.example.public_ip
}
```

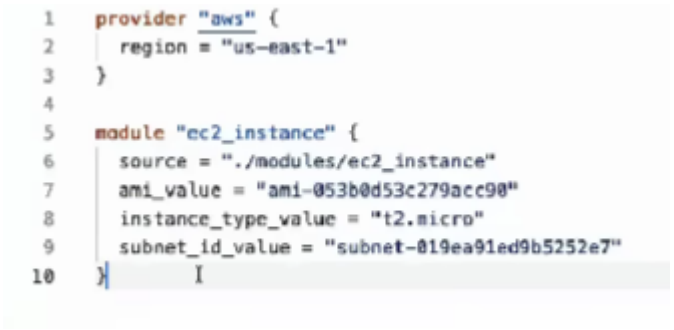
Call the Module in Your Main Configuration :

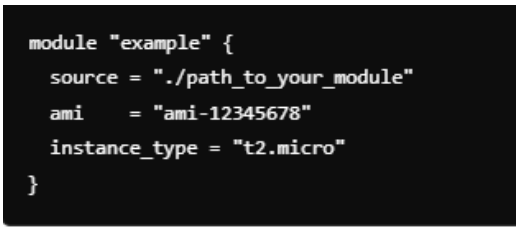
A module is typically a directory with Terraform configuration files. For example:

Example Module Structure (`modules/ec2_instance`):



In your root Terraform configuration, use the `module` block to call the module. You need to specify the `source` (where Terraform can find the module) and provide any required input variables.





Terraform State

What is Terraform State : Terraform state is a file that keeps track of the current state of your infrastructure as defined by your Terraform configuration files. This state file, typically named ***"terraform.tfstate"***, contains information about the resources you've created, their current status, and any dependencies between them.

Importance of Terraform State file :

1. **Resource Tracking**: Terraform state files keep track of all the resources managed by Terraform, including their IDs and attributes. This helps Terraform understand what is currently deployed and how it maps to the configuration files.
2. **Change Management**: When you run commands like `terraform apply`, Terraform compares the desired state (from your configuration files) to the actual state (from the state file). It then determines what changes need to be made to align the actual state with the desired state.
3. **Efficient Operations**: By maintaining a state file, Terraform can perform operations more efficiently. It avoids querying the cloud provider or other services for information about existing resources, which speeds up operations.
4. **Collaboration**: In a team environment, the state file can be stored remotely (e.g., in an S3 bucket, Azure Blob Storage, etc.) to allow multiple people to work on the same infrastructure and ensure they are working with the same state.
5. **Resource Metadata**: The state file contains metadata about resources, which is used by Terraform to manage dependencies and understand how resources are related.

State files are often stored locally by default, but in a collaborative setting or in production environments, they are usually stored in a remote backend for better reliability and consistency.

Drawbacks of Terraform State file :

1. **Security Risks**: It might contain sensitive data like resource IDs or secrets, which can be risky if not protected.
2. **Corruption Risk**: If the state file gets corrupted, it can cause problems with managing your resources.
3. **Complex to Manage**: Making manual changes to the state file can be tricky and error-prone.
4. **Sync Issues**: When using a remote state file, multiple users or processes might cause conflicts if not properly synchronized.
5. **Performance Problems**: Very large state files can slow down Terraform operations.
6. **Version Control**: Tracking changes and recovering from mistakes can be difficult if the state file isn't properly backed up or versioned.
7. **Backup Needs**: Without proper backups, recovering from accidental deletions or corruption can be hard.

How to Overcome the Drawbacks of Terraform State file :

Certainly! Here are some ways to mitigate the drawbacks of the Terraform state file:

1. **Use Remote State Storage:** A ***remote backend*** stores the Terraform state file outside of your local file system and version control. Store your state file in a remote backend like Amazon S3, Azure Blob Storage, or Google Cloud Storage. This helps

ensure that the state file is always available and can be accessed by multiple team members. Additionally, using a remote backend with ***state locking*** (e.g., DynamoDB for AWS) prevents conflicts by ensuring that only one person can make changes at a time.

2. **Encrypt the State File:** To protect sensitive data, always enable encryption for your state file. Most remote backends support encryption at rest and in transit. For example, you can enable server-side encryption for S3 buckets.
3. **Use Version Control:** Keep your state file under version control. This allows you to track changes and revert to previous versions if something goes wrong. Some remote backends automatically version your state file.
4. **Implement Access Controls:** Restrict access to the state file to only those who need it. Use IAM roles and policies to control who can read or modify the state file.
5. **Regular Backups:** Regularly back up your state file to ensure you can recover it in case of corruption or loss. Automated backup solutions can help with this.
6. **Monitor and Audit:** Set up monitoring and auditing for your state file. This can help you detect unauthorized changes or access and take corrective actions promptly.

How is Terraform State Managed?

- **Local State**: By default, Terraform stores the state file locally in the same directory where Terraform is run.
- **Remote State**: For better collaboration and security, it's recommended to store the state file in a remote backend like AWS S3, Azure Blob Storage, or Terraform Cloud. This allows for versioning, encryption and secure sharing.

State locking in Terraform:

Terraform state locking is a feature that helps prevent multiple people from making changes to your infrastructure at the same time, which could cause conflicts or corruption.

What it does: When you run commands (like `terraform apply` or `terraform plan`), that change your infrastructure, Terraform locks the state file. This means no one else can make changes until the lock is released.

Why it's important: It ensures that only one set of changes is applied at a time, keeping your infrastructure consistent and preventing errors.

How it works: The lock is automatically applied when you run commands (like terraform apply or terraform plan). If someone else tries to run these commands while the state is locked, they'll have to wait until the lock is released.

Unlocking: The lock is usually released automatically when the command finishes. If something goes wrong and the lock isn't released, you can manually unlock it using the ***"terraform force-unlock"*** command.

Configure Terraform to Use the S3 Backend with DynamoDB for Locking:

In your Terraform configuration, you will specify the S3 backend to store the state file and optionally use DynamoDB for locking.

Example Configuration

main.tf file

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

resource "aws_instance" "abhishek" {
  instance_type = "t2.micro"
  ami = "ami-053b0d53c279acc90" # change this
  subnet_id = "subnet-019ea91ed9b5252e7" # change this
}

resource "aws_s3_bucket" "s3_bucket" {
  bucket = "abhishek-s3-demo-xyz" # change this
}

resource "aws_dynamodb_table" "terraform_lock" {
  name           = "terraform-lock"
  billing_mode   = "PAY_PER_REQUEST"
  hash_key       = "LockID"

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

backend.tf file

```
terraform {
  backend "s3" {
    bucket         = "your-s3-bucket"          # Replace with your S3 bucket name
    key            = "terraform/state.tfstate" # Path to the state file within the bucket
    region         = "us-east-1"               # Replace with your AWS region
    dynamodb_table = "terraform-locks"         # Replace with your DynamoDB table name
    encrypt        = true                      # Optional: Enable server-side encryption f
  }
}
```

```
terraform {
  backend "s3" {
    bucket         = "abhishek-s3-demo-xyz" # change this
    key            = "abhi/terraform.tfstate"
    region         = "us-east-1"
    encrypt        = true
    dynamodb_table = "terraform-lock"
  }
}
```

PROVISIONERS in Terraform

Provisioners in Terraform are used to execute scripts on a local or remote machine during the creation or destruction of resources. They are particularly useful for tasks that cannot be accomplished with Terraform's declarative syntax alone

Types of Provisioners

1. **`local-exec` Provisioner:** Executes a command or script on the machine where Terraform is running.

```
resource "null_resource" "example" {
  provisioner "local-exec" {
    command = "echo Hello, World!"
  }
}
```

`null_resource`: This is a special type of resource in Terraform that doesn't actually create any infrastructure. Instead, it can be used to execute provisioners or manage dependencies.

`"example"`: This is the name of the resource. It's a reference you can use to refer to this resource within your Terraform configuration.

`local-exec`: This provisioner runs commands on the machine where Terraform is being executed, not on the target infrastructure (e.g., EC2 instances).

`command = "echo Hello, World!"`: This command will be executed by the `local-exec` provisioner. In this case, it simply prints "Hello, World!" to the console or terminal where Terraform is run.

What This Script Does

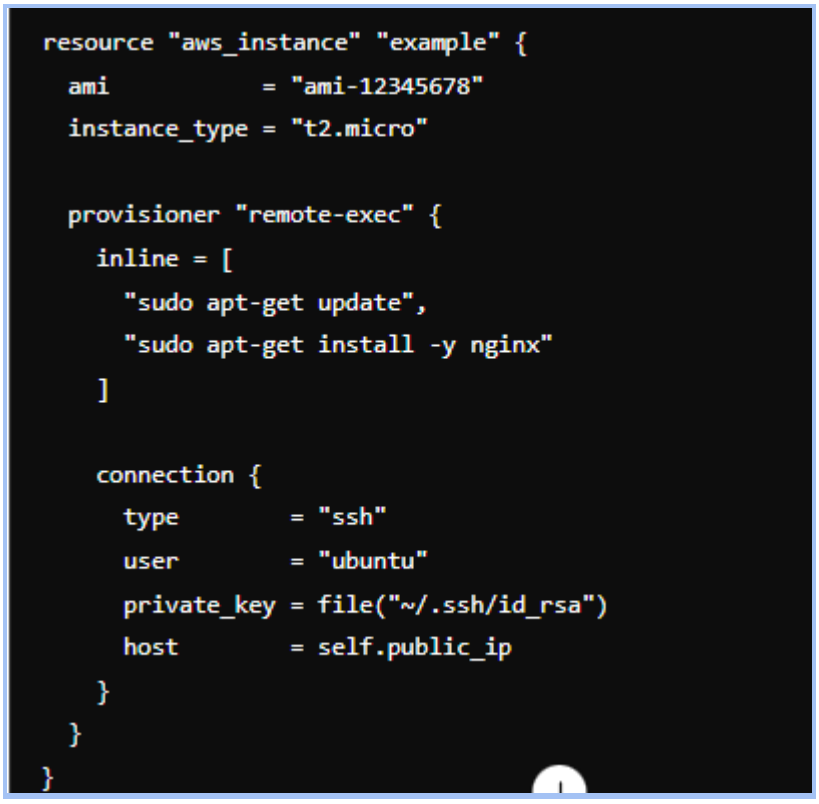
- When you run `terraform apply`, Terraform will create the `null_resource` and execute the `local-exec` provisioner.
- The `local-exec` provisioner will run the command `echo Hello, World!` on your local machine (the machine where Terraform is executed).
- As a result, you will see "Hello, World!" printed to the terminal or console.

2. `remote-exec` Provisioner:

Executes a command or script on the remote resource being created.

Requires SSH or WinRM access to the target machine.

Useful for configuring instances after they are provisioned, such as installing software or setting up services.



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

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

    connection {
      type        = "ssh"
      user        = "ubuntu"
      private_key = file("~/.ssh/id_rsa")
      host        = self.public_ip
    }
  }
}
```

`ami`: Specifies which pre-configured image to use for the instance. Replace `"ami-12345678"` with the ID of an actual AMI in your region.

`instance_type`: Defines the size and capacity of the instance. `"t2.micro"` is a small, inexpensive instance type.

`remote-exec`: A provisioner that runs commands on the EC2 instance after it has been created.

`inline`: A list of commands to run on the EC2 instance. Here, it:

- Updates the package list with `sudo apt-get update`.
- Installs Nginx, a web server, with `sudo apt-get install -y nginx`.

`type`: Specifies that the connection will be made using SSH.

`user`: The username for SSH access (e.g., `"ubuntu"` for Ubuntu-based instances).

`private_key`: The path to the SSH private key file used to authenticate the SSH session.

`host`: The public IP address of the EC2 instance. `self.public_ip` refers to the instance's public IP that Terraform provides.

When you run `terraform apply`:

1. Terraform will create an EC2 instance using the specified AMI and instance type.
2. Once the instance is up and running, Terraform will connect to it via SSH.
3. Terraform will then run the commands to update the package list and install Nginx on the instance.

3. `file` Provisioner: The file provisioner is used to copy files or directories from the local machine to a remote machine. This is useful for deploying configuration files, scripts, or other assets to a provisioned instance.

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

  provisioner "file" {
    source      = "local-file.txt"
    destination = "/home/ubuntu/remote-file.txt"

    connection {
      type        = "ssh"
      user        = "ubuntu"
      private_key = file("~/.ssh/id_rsa")
      host        = self.public_ip
    }
  }
}
```

`ami`: Specifies which pre-configured image to use. Replace `"ami-12345678"` with the ID of a real AMI.

`instance_type`: Defines the instance size. `"t2.micro"` is a basic, inexpensive type.

`file`: This provisioner copies a file from your local machine to the remote EC2 instance.

- **`source`**: The path to the file on your local machine (`"local-file.txt"`).
- **`destination`**: The path on the EC2 instance where the file will be placed (`"/home/ubuntu/remote-file.txt"`).

`connection` Block: Defines how to connect to the EC2 instance via SSH.

    - **`type`**: Specifies that SSH is used.
    - **`user`**: The username to use for SSH (`"ubuntu"` in this case, which is common for Ubuntu AMIs).
    - **`private_key`**: Path to your SSH private key file, used to authenticate the SSH session.
    - **`host`**: The EC2 instance's public IP address (`self.public_ip` gets this IP from Terraform).

When you run `terraform apply`:

1. Terraform creates the EC2 instance using the specified AMI and instance type.
2. Once the instance is up, Terraform connects to it via SSH.
3. It then copies the file `"local-file.txt"` from your local machine to `"/home/ubuntu/remote-file.txt"` on the EC2 instance.

This script automates the process of transferring files to your EC2 instance as part of the infrastructure setup.

Terraform Workspaces

What is Terraform workspaces?

A Terraform workspace is a feature that allows you to manage multiple environments within a single Terraform configuration. Each workspace is a separate instance of your infrastructure, with its own state file and variables. This means you can have different environments like development, staging, and production, all managed from the same configuration but isolated from each other.

Benefits of Terraform Workspace :

1. **State Isolation**: Each workspace maintains its own state file. The state file keeps track of the resources that Terraform manages. By using workspaces, you can isolate

the state for different environments (like development, staging, and production) or different versions of the same environment.

2. **Multiple Environments**: Instead of creating separate Terraform configurations for each environment, you can use workspaces to keep everything within one configuration. For example, you could have a `dev` workspace for your development environment and a `prod` workspace for your production environment.
3. **Commands**: Terraform provides commands to create, select, and manage workspaces. For example:
    - `terraform workspace new <workspace_name>`: Creates a new workspace.
    - `terraform workspace select <workspace_name>`: Switches to the specified workspace.
    - `terraform workspace list`: Lists all available workspaces.
4. **Usage**: Workspaces are useful for scenarios where you need to manage similar infrastructure in different contexts but want to keep their configurations separate.

******** Example of terraform workspace to create ec2 instance with different instance types under the variable block for dev, stage and prod environment -***

Define Variables (`variables.tf`) :

Define a variable for the instance type and any other parameters you might need.

```
variable "instance_type" {
  description = "The type of EC2 instance to create"
  type        = string
}
```

Create the Main Configuration (`main.tf`)

Use the variable for the instance type and include a tag to identify the workspace.

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

resource "aws_instance" "example" {
  ami           = "ami-0c55b159cbfafe1f0" # Replace with your AMI ID
  instance_type = var.instance_type
}
```

Define Workspace-Specific Values (`terraform.tfvars`)

You need different values for each workspace. Create separate `terraform.tfvars` files for each environment. For example:

`dev.tfvars`:

```hcl
instance_type = "t2.micro"
```

`stage.tfvars`:

```hcl
instance_type = "t2.medium"
```

`prod.tfvars`:

```hcl
instance_type = "t3.large"
```

Initialize Terraform

Run `terraform init` to initialize your Terraform setup.

Create and Switch Workspaces

```
# Create workspaces
terraform workspace new dev
terraform workspace new stage
terraform workspace new prod

# Switch to the dev workspace
terraform workspace select dev
```

Apply Configuration with the Appropriate Variables

When applying the configuration, specify the terraform.tfvars file that corresponds to the selected workspace.

```
# Apply configuration for the dev environment
terraform workspace select dev
terraform apply -var-file="dev.tfvars" -auto-approve

# Apply configuration for the stage environment
terraform workspace select stage
terraform apply -var-file="stage.tfvars" -auto-approve

# Apply configuration for the prod environment
terraform workspace select prod
terraform apply -var-file="prod.tfvars" -auto-approve
```

(OR)

Create a module for creating EC2 instance - this module can be used to create ec2 instance in any environment.

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

variable "ami" {
  description = "This is AMI for the instance"
}

variable "instance_type" {
  description = "This is the instance type, for example: t2.micro"
}

resource "aws_instance" "example" {
    ami = var.ami
    instance_type = var.instance_type
}
```

We can have “terraform.tfvars” file for passing ami value :

```
ami = "ami-053b0d53c279acc90"
```

Using the above module, and passing the instance type value for each environment (dev, stage, prod) in the “main.tf” file itself. Not required to have different “terraform.tfvars” file for passing instance type values.

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

variable "ami" {
  description = "value"
}

variable "instance_type" {
  description = "value"
  type = map(string)

  default = {
    "dev" = "t2.micro"
    "stage" = "t2.medium"
    "prod" = "t2.xlarge"
  }
}

module "ec2_instance" {
  source = "./modules/ec2_instance"
  ami = var.ami
  instance_type = lookup(var.instance_type, terraform.workspace, "t2.micro")
}
```

lookup(var.instance_type, terraform.workspace, "t2.micro"): This function looks up the value of the `instance_type` map using the current workspace name (`terraform.workspace`) as the key. If the key does not exist, it defaults to `"t2.micro"`. For instance:

- In the `dev` workspace, `lookup(var.instance_type, "dev", "t2.micro")` returns `"t2.micro"`.
- In the `stage` workspace, `lookup(var.instance_type, "stage", "t2.micro")` returns `"t2.medium"`.
- In the `prod` workspace, `lookup(var.instance_type, "prod", "t2.micro")` returns `"t2.xlarge"`.

In the terminal, run the commands :

Apply configuration for the dev environment

```
terraform workspace select dev
terraform apply
```

Apply configuration for the stage environment

```
terraform workspace select stage
terraform apply
```

Apply configuration for the prod environment

```
terraform workspace select prod
terraform apply
```

Terraform Secret Management - Hashicorp Vault Integration

What is HashiCorp Vault?

Vault is a tool for securely storing and managing sensitive information like passwords, API keys, or database credentials. When you integrate Vault with Terraform, you can safely retrieve secrets and use them in your infrastructure without exposing them in your configuration files.

Steps to Manage Secrets in Terraform Using Vault

1. **Store the secret in Vault.**
2. **Use the Vault provider in Terraform.**
3. **Fetch the secret from Vault and use it in Terraform resources.**

Example: Fetching a Database Password from Vault

1. Store the Secret in Vault

Let's say you have a database password you want to store securely in Vault.

Run this command in Vault to store the secret:

```
vault kv put secret/db-pass password="MySecurePassword123"
```

Here,

`Vault`: This is the command-line interface (CLI) tool for interacting with **HashiCorp Vault**.

`kv:` This stands for "Key-Value" secret engine, which is one of the storage engines provided by Vault.

`put`: The `put` command tells Vault to store data at a specific path. This stores the password `MySecurePassword123` under the path `secret/db-pass` in Vault.

`secret/db-pass`: This is the **path** in Vault where the secret is stored. Think of it as a folder structure where you store your secrets.

`password="MySecurePassword123"`: This is the **key-value pair** being stored in Vault. The key is `password`, and the value is `"MySecurePassword123"`.

2. Configure Vault Provider in Terraform

Now, you need to set up Terraform to connect to Vault and retrieve the secret. First, you configure the Vault provider in Terraform.

```
provider "vault" {
  address = "http://127.0.0.1:8200"  # Vault's address
  token   = "s.xxxxxxxx"             # Replace with your Vault token
}
```

This allow Terraform to connect to **HashiCorp Vault** and retrieve secrets.

Here,

`provider "vault"`: This tells Terraform that you're using the **Vault provider**.

`address = "http://127.0.0.1:8200"`: This is the **URL of the Vault server**.

`token = "s.xxxxxxxx"`: This is the **authentication token** used to access Vault.

3. Fetch the Secret in Terraform and Use It

Next, you tell Terraform to retrieve the secret from Vault and use it in your infrastructure, like setting up a database.

```
data "vault_generic_secret" "db_password" {
  path = "secret/db-pass"
}

resource "aws_db_instance" "my_database" {
  identifier         = "mydb"
  engine             = "mysql"
  instance_class     = "db.t2.micro"
  allocated_storage  = 20
  name               = "mydb"
  username           = "admin"
  password           = data.vault_generic_secret.db_password.data["password"]
}
```

Here,

`data "vault_generic_secret" "db_password":` This part is called a **data source**. It tells Terraform to retrieve a secret from HashiCorp Vault.

`path = "secret/db-pass":` This specifies the **path** in Vault where the secret is stored. In this case, the path is secret/db-pass.

`resource "aws_db_instance" "my_database":` This part defines a resource for creating an AWS RDS database instance.

`identifier = "mydb":` This sets a unique name for the database instance.

`engine = "mysql":` Specifies that the database engine is MySQL.

`instance_class = "db.t2.micro":` Defines the type of database instance to use.

`allocated_storage = 20:` Sets the storage size for the database in gigabytes.

`name = "mydb":` The name of the database.

`username = "admin":` The username for accessing the database.

In this example:

- **Vault Secret Fetching:** The `data "vault_generic_secret"` block fetches the password from Vault.
- **Using the Secret:** The `aws_db_instance` resource uses that password when creating a new database instance.

4. Apply Terraform

Once everything is set up, run these commands to apply the changes and provision your infrastructure with the secret fetched securely from Vault.

```
terraform init
terraform apply
```

Key Benefits:

- **Secure Secrets Management:** Secrets are not hardcoded in Terraform files.
- **Dynamic Secrets:** Vault can generate secrets dynamically and rotate them without exposing sensitive information.

This way, you keep your sensitive data secure while using it in Terraform configurations.