**Terraform**Terraform is an open-source infrastructure as code (IaC) tool developed by hashicorp that lets us automate the provisioning, configuration and management of the infrastructure across multiple cloud platforms and on-prem environments.

**Commands:**

1. **terraform init**  
   Initializes the backend (if we defined remote backend like s3, terraform cloud otherwise defaults to local backend)  
   Installs/Downloads provider plugins from terraform registry and stores in a “. terraform” directory.
2. **terraform plan**  
   It generates the execution plan:  
   What resource will be created  
   What resource will be changed  
   What resource will be destroyed  
   Why??  
   To review changes before applying  
   To detect mistakes in your configurations.  
   To confirm what changes will be made by ‘terraform apply’ command.
3. **terraform apply**  
   terraform apply executes the plan created by terraform and makes real changes on the cloud infrastructure and changes is based on the .tf configuration files. Based on the plan creates, updates & destroys resources in the cloud infrastructure.
4. **terraform destroy**terraform destroy is used to delete or tear down all the infrastructure managed by your current terraform project.  
   reads .tf files to understand the resources it manages  
   checks state file (terraform.tfstate) to know what infrastructure currently exists.  
   Send API calls to the AWS to delete all resources created as per .tf  
   Removes infrastructure from state file once after successfully destroyed.

**provider block:**provider block is a plugin that helps terraform to understand where to create infrastructure in the particular cloud platform.

**Multiple-regions:**provider “aws” {  
alias = “us-east-1”  
region = “us-east-1”  
}

provider “aws” {  
alias = “us-west-2”  
region = “us-west-2”  
}

resource “aws\_instance” “Example1” {  
ami = “”  
instance\_type = “t2.micro”  
key\_name = "Windows-EC2Connect"  
provider = “aws.us-east-1”  
}

resource “aws\_instance” “Example2” {  
ami = “”  
instance\_type = “t2.micro”  
key\_name = "Windows-EC2Connect"  
provider = “aws.us-west-2”  
}

**Variables:**Input and output variables in Terraform are essential for parameterizing and sharing values within your Terraform configurations and modules. They allow you to make your configurations more dynamic, reusable, and flexible.

**Input Variables:**Input variables are used to parameterize your Terraform configurations. They allow you to pass values into your modules or configurations from the outside. Input variables can be defined within a module or at the root level of your configuration. Here's how you define an input variable:  
variable "example\_var" {  
 description = "An example input variable"  
 type = string  
 default = "default\_value"  
}

In this example:

* **variable** is used to declare an input variable named example\_var.
* **description** provides a human-readable description of the variable.
* **type** specifies the data type of the variable (e.g., string, number, list, map, etc.).
* **default** provides a default value for the variable, which is optional.

You can then use the input variable within your module or configuration like this:  
resource "example\_resource" "example" {

name = var.example\_var

# other resource configurations

}  
You reference the input variable using var.example\_var.

**Output variables:**  
Output variables allow you to expose values from your module or configuration, making them available for use in other parts of your Terraform setup. Here's how you define an output variable:  
output "example\_output" {

description = "An example output variable"

value = resource.example\_resource.example.id

}

In this example:

* output is used to declare an output variable named example\_output.
* description provides a description of the output variable.
* value specifies the value that you want to expose as an output variable. This value can be a resource attribute, a computed value, or any other expression.

You can reference output variables in the root module or in other modules by using the syntax module.module\_name.output\_name, where module\_name is the name of the module containing the output variable.

For example, if you have an output variable named example\_output in a module called example\_module, you can access it in the root module like this:  
output "root\_output" {

value = module.example\_module.example\_output

}

This allows you to share data and values between different parts of your Terraform configuration and create more modular and maintainable infrastructure-as-code setups.

**Example:**  
# Define an input variable for the EC2 instance type

variable "instance\_type" {

description = "EC2 instance type"

type = string

default = "t2.micro"

}

# Define an input variable for the EC2 instance AMI ID

variable "ami\_id" {

description = "EC2 AMI ID"

type = string

}

# Configure the AWS provider using the input variables

provider "aws" {

region = "us-east-1"

}

# Create an EC2 instance using the input variables

resource "aws\_instance" "example\_instance" {

ami = var.ami\_id

instance\_type = var.instance\_type

}

# Define an output variable to expose the public IP address of the EC2 instance

output "public\_ip" {

description = "Public IP address of the EC2 instance"

value = aws\_instance.example\_instance.public\_ip

}

**Terraform tfvars**  
In Terraform, .tfvars files (typically with a .tfvars extension) are used to set specific values for input variables defined in your Terraform configuration.

They allow you to separate configuration values from your Terraform code, making it easier to manage different configurations for different environments (e.g., development, staging, production) or to store sensitive information without exposing it in your code.

Here's the purpose of .tfvars files:  
**Separation of Configuration from Code**: Input variables in Terraform are meant to be configurable so that you can use the same code with different sets of values. Instead of hardcoding these values directly into your .tf files, you use .tfvars files to keep the configuration separate. This makes it easier to maintain and manage configurations for different environments.

**Sensitive Information**: .tfvars files are a common place to store sensitive information like API keys, access credentials, or secrets. These sensitive values can be kept outside the version control system, enhancing security and preventing accidental exposure of secrets in your codebase.

**Reusability**: By keeping configuration values in separate .tfvars files, you can reuse the same Terraform code with different sets of variables. This is useful for creating infrastructure for different projects or environments using a single set of Terraform modules.

**Collaboration**: When working in a team, each team member can have their own .tfvars file to set values specific to their environment or workflow. This avoids conflicts in the codebase when multiple people are working on the same Terraform project.

**Summary**

Here's how you typically use .tfvars files

1. Define your input variables in your Terraform code (e.g., in a variables.tf file).
2. Create one or more .tfvars files, each containing specific values for those input variables.
3. When running Terraform commands (e.g., terraform apply, terraform plan), you can specify which .tfvars file(s) to use with the -var-file option:

terraform apply -var-file=dev.tfvars

By using .tfvars files, you can keep your Terraform code more generic and flexible while tailoring configurations to different scenarios and environments.

**Conditional Expressions**

Conditional expressions in Terraform are used to define conditional logic within your configurations. They allow you to make decisions or set values based on conditions. Conditional expressions are typically used to control whether resources are created or configured based on the evaluation of a condition.

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.

Here are some common use cases and examples of how to use conditional expressions in Terraform:

**Conditional Resource Creation Example**

resource "aws\_instance" "example" {

count = var.create\_instance ? 1 : 0

ami = "ami-XXXXXXXXXXXXXXXXX"

instance\_type = "t2.micro"

}

In this example, the count attribute of the aws\_instance resource uses a conditional expression. If the create\_instance variable is true, it creates one EC2 instance. If create\_instance is false, it creates zero instances, effectively skipping resource creation.

**Conditional Variable Assignment Example**

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]

}

}

In this example, the locals block uses a conditional expression to assign a value to the subnet\_cidr local variable based on the value of the environment variable. If environment is set to "production", it uses the production\_subnet\_cidr variable; otherwise, it uses the development\_subnet\_cidr variable.

**Conditional Resource Configuration**

resource "aws\_security\_group" "example" {

name = "example-sg"

description = "Example security group"

ingress {

from\_port = 22

to\_port = 22

protocol = "tcp"

cidr\_blocks = var.enable\_ssh ? ["0.0.0.0/0"] : []

}

}

In this example, the ingress block within the aws\_security\_group resource uses a conditional expression to control whether SSH access is allowed. If enable\_ssh is true, it allows SSH traffic from any source ("0.0.0.0/0"); otherwise, it allows no inbound traffic.

Conditional expressions in Terraform provide a powerful way to make decisions and customize your infrastructure deployments based on various conditions and variables. They enhance the flexibility and reusability of your Terraform configurations.

**Built-in Functions**

Terraform is an infrastructure as code (IaC) tool that allows you to define and provision infrastructure resources in a declarative manner. Terraform provides a wide range of built-in functions that you can use within your configuration files (usually written in HashiCorp Configuration Language, or HCL) to manipulate and transform data. These functions help you perform various tasks when defining your infrastructure. Here are some commonly used built-in functions in Terraform:

1. concat(list1, list2, ...): Combines multiple lists into a single list.

variable "list1" {

type = list

default = ["a", "b"]

}

variable "list2" {

type = list

default = ["c", "d"]

}

output "combined\_list" {

value = concat(var.list1, var.list2)

}

1. element(list, index): Returns the element at the specified index in a list.

variable "my\_list" {

type = list

default = ["apple", "banana", "cherry"]

}

output "selected\_element" {

value = element(var.my\_list, 1) # Returns "banana"

}

1. length(list): Returns the number of elements in a list.

variable "my\_list" {

type = list

default = ["apple", "banana", "cherry"]

}

output "list\_length" {

value = length(var.my\_list) # Returns 3

}

1. map(key, value): Creates a map from a list of keys and a list of values.

variable "keys" {

type = list

default = ["name", "age"]

}

variable "values" {

type = list

default = ["Alice", 30]

}

output "my\_map" {

value = map(var.keys, var.values) # Returns {"name" = "Alice", "age" = 30}

}

1. lookup(map, key): Retrieves the value associated with a specific key in a map.

variable "my\_map" {

type = map(string)

default = {"name" = "Alice", "age" = "30"}

}

output "value" {

value = lookup(var.my\_map, "name") # Returns "Alice"

}

1. join(separator, list): Joins the elements of a list into a single string using the specified separator.

variable "my\_list" {

type = list

default = ["apple", "banana", "cherry"]

}

output "joined\_string" {

value = join(", ", var.my\_list) # Returns "apple, banana, cherry"

}

These are just a few examples of the built-in functions available in Terraform. You can find more functions and detailed documentation in the official Terraform documentation, which is regularly updated to include new features and improvements.

**Modules**

The advantage of using Terraform modules in your infrastructure as code (IaC) projects lies in improved organization, reusability, and maintainability. Here are the key benefits:

1. **Modularity**: Terraform modules allow you to break down your infrastructure configuration into smaller, self-contained components. This modularity makes it easier to manage and reason about your infrastructure because each module handles a specific piece of functionality, such as an EC2 instance, a database, or a network configuration.
2. **Reusability**: With modules, you can create reusable templates for common infrastructure components. Instead of rewriting similar configurations for multiple projects, you can reuse modules across different Terraform projects. This reduces duplication and promotes consistency in your infrastructure.
3. **Simplified Collaboration**: Modules make it easier for teams to collaborate on infrastructure projects. Different team members can work on separate modules independently, and then these modules can be combined to build complex infrastructure deployments. This division of labor can streamline development and reduce conflicts in the codebase.
4. **Versioning and Maintenance**: Modules can have their own versioning, making it easier to manage updates and changes. When you update a module, you can increment its version, and other projects using that module can choose when to adopt the new version, helping to prevent unexpected changes in existing deployments.
5. **Abstraction**: Modules can abstract away the complexity of underlying resources. For example, an EC2 instance module can hide the details of security groups, subnets, and other configurations, allowing users to focus on high-level parameters like instance type and image ID.
6. **Testing and Validation**: Modules can be individually tested and validated, ensuring that they work correctly before being used in multiple projects. This reduces the risk of errors propagating across your infrastructure.
7. **Documentation**: Modules promote self-documentation. When you define variables, outputs, and resource dependencies within a module, it becomes clear how the module should be used, making it easier for others (or your future self) to understand and work with.
8. **Scalability**: As your infrastructure grows, modules provide a scalable approach to managing complexity. You can continue to create new modules for different components of your architecture, maintaining a clean and organized codebase.
9. **Security and Compliance**: Modules can encapsulate security and compliance best practices. For instance, you can create a module for launching EC2 instances with predefined security groups, IAM roles, and other security-related configurations, ensuring consistency and compliance across your deployments.

**Terraform stateFile:**It records the infrastructure that has been created/modified by terraform apply command. Heart of terraform.  
While modifying, terraform apply is going to check/compare the stateFile to understand what is the infrastructure that has been already created and what is the new infrastructure to create.

**Drawbacks:**

1. stateFile captures everything including sensitive information. If anyone got access to the system which has stateFile. Sensitive information will be breached.
2. If someone misses pushing the updated stateFile to VCS, new configuration changes will be missed.

We can resolve above drawbacks by implementing remote backend like s3, terraform cloud etc.,  
We can host stateFile in s3 bucket and restrict outsiders to access this. Also automatically updates the stateFile as we have hosted it in external service s3 bucket.

terraform show => it checks whether stateFile exists or not. if not displays “No state”.

**Overcoming Disadvantages with Remote Backends (e.g., S3):**

A remote backend stores the Terraform state file outside of your local file system and version control. Using S3 as a remote backend is a popular choice due to its reliability and scalability. Here's how to set it up:

1. **Create an S3 Bucket**: Create an S3 bucket in your AWS account to store the Terraform state. Ensure that the appropriate IAM permissions are set up.
2. **Configure Remote Backend in Terraform:**

# In your Terraform configuration file (e.g., main.tf), define the remote backend.

terraform {

backend "s3" {

bucket = "your-terraform-state-bucket"

key = "path/to/your/terraform.tfstate"

region = "us-east-1" # Change to your desired region

encrypt = true

dynamodb\_table = "your-dynamodb-table"

}

}

Replace "your-terraform-state-bucket" and "path/to/your/terraform.tfstate" with your S3 bucket and desired state file path.

**DynamoDB Table for State Locking:**

To enable state locking, create a DynamoDB table and provide its name in the dynamodb\_table field. This prevents concurrent access issues when multiple users or processes run Terraform.

**State Locking with DynamoDB:**

DynamoDB is used for state locking when a remote backend is configured. It ensures that only one user or process can modify the Terraform state at a time. Here's how to create a DynamoDB table and configure it for state locking:

1. **Create a DynamoDB Table:**

You can create a DynamoDB table using the AWS Management Console or AWS CLI. Here's an AWS CLI example:

aws dynamodb create-table --table-name your-dynamodb-table --attribute-definitions AttributeName=LockID,AttributeType=S --key-schema AttributeName=LockID,KeyType=HASH --provisioned-throughput ReadCapacityUnits=5,WriteCapacityUnits=5

1. **Configure the DynamoDB Table in Terraform Backend Configuration:**

In your Terraform configuration, as shown above, provide the DynamoDB table name in the dynamodb\_table field under the backend configuration.

By following these steps, you can securely store your Terraform state in S3 with state locking using DynamoDB, mitigating the disadvantages of storing sensitive information in version control systems and ensuring safe concurrent access to your infrastructure. For a complete example in Markdown format, you can refer to the provided example below:

**## Configure Terraform Remote Backend**

Example:

**main.tf**

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**

terraform {

backend "s3" {

bucket = "abhishek-s3-demo-xyz" # change this

key = "abhi/terraform.tfstate"

region = "us-east-1"

encrypt = true

dynamodb\_table = "terraform-lock"

}

}

**Provisioner:**

In Terraform, a provisioner is used to execute scripts or commands on a local or remote resource after it has been created.

**Types of Provisioners in Terraform**

1. **local-exec**: Runs a command **on your local machine** (where Terraform is running).

Example:

resource "aws\_instance" "example" {

ami = "ami-12345678"

instance\_type = "t2.micro"

provisioner "local-exec" {

command = "echo Instance created: ${self.public\_ip}"

}

}

1. **remote-exec**: Runs commands **on the provisioned resource** (like an EC2 instance).

Example:  
resource "aws\_instance" "server" {

  ami                    = "ami-0e35ddab05955cf57"

  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 apt install python3-flask",

      "sudo python3 app.py &",

    ]

  }

}

1. **file**: Copies files or content **from local to remote** machine.

Example:  
resource "aws\_instance" "server" {

  ami                    = "ami-0e35ddab05955cf57"

  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

  }

**Terraform workspace:**

Instead of rewriting same terraform project for different environments, we can go for terraform workspace which helps to select different environments and create infrastructure in that environment workspace.

Commands:

terraform workspace select dev => will login to the specified workspace.

terraform workspace select staging

terraform workspace select prod

terraform workspace show => will display the current logged in workspace.

tree => will display the structure of project.