1. What is Terraform and its primary use?

Terraform, by HashiCorp, is an open-source infrastructure as code (IaC) tool. It manages cloud resources using a declarative language for automated, consistent deployments across multiple providers.

2. How to install Terraform?

Download Terraform for your OS from the official site. Move the binary to a directory in your PATH. Verify installation with terraform --version.

1.1.9

3. What are Terraform Providers?

Providers are plugins enabling Terraform to manage external APIs (e.g., AWS, Azure). They define and manage resource lifecycles across platforms.

4. Explain a Terraform configuration file's structure.

Written in HCL, it includes:

provider: Specifies the cloud platform.

resource: Defines infrastructure components.

variable: For input variables.

output: Displays results.

5. What is a Terraform state file?

The state file tracks managed infrastructure, ensuring changes align with the defined configuration.

6. How to initialize a Terraform configuration?

Run terraform init in your config directory. It downloads providers and sets up the working directory.

7. Purpose of the terraform plan command?

Generates an execution plan outlining changes Terraform will make, helping to review updates before applying.

8. How to apply a Terraform configuration?

Run terraform init, review with terraform plan, and execute changes using terraform apply.

9. Use of terraform destroy command?

Removes all resources defined in the configuration to clean up infrastructure.

10. What is a "resource" in Terraform?

A resource is a manageable infrastructure component (e.g., VM, DNS record).

11. Competitors of Terraform?

Ansible: Automation and agentless design.

AWS CloudFormation: AWS-native IaC tool.

Google Cloud Deployment Manager: For Google Cloud environments.

Azure Resource Manager (ARM): Native for Azure.

12. How does Terraform Core operate?

Terraform Core processes HCL files, builds a dependency tree, applies infrastructure changes, and updates the state file.

13. What is "Terraform D"?

"Terraform D" likely refers to Terraform versions or iterations for IaC management.

14. What does IaC mean?

Infrastructure as Code (IaC) uses code to manage infrastructure instead of manual setups.

15. Why use Terraform in DevOps?

Simplifies IaC.

Automates provisioning.

Ensures consistency and scalability.

16. What is null_resource in Terraform?

Used for tasks without directly creating resources, enabling triggers or local commands.

17. How to manage multiple environments in Terraform?

Use separate state files, directories, or workspaces, and parameterize configs with variables.

18. What are Terraform Modules?

Reusable configuration units simplify complex setups and improve code maintainability.

19. Difference between terraform apply and terraform plan -out?

terraform plan: Generates a plan.

terraform apply: Executes the plan.

plan -out: Saves the plan for later use with apply.

20. What are remote backends?

Remote backends store state files in shared locations (e.g., S3), ensuring consistency and collaboration.

21. How to handle secrets in Terraform?

Use Vault, AWS Secrets Manager, or environment variables to avoid hardcoding secrets.

22. Use of terraform taint command?

Marks a resource for destruction and recreation in the next apply.

23. Provisioning AWS infrastructure with Terraform?

Write HCL files, initialize with terraform init, preview with plan, and deploy with apply.

24. Purpose of terraform validate command?

Checks syntax and configuration errors before deployment.

25. Difference between count and for_each?

count: Repeats resources numerically.

for_each: Iterates over sets/maps for dynamic resource creation.

26. How to import existing infrastructure?

Use terraform import <resource_type> <ID> to link Terraform to current infrastructure.

27. What is a Private Module Registry?

A secure repository for sharing reusable Terraform modules within an organization.

28. Can Terraform manage on-premise infrastructure?

Yes, using providers like VMware or custom scripts.

29. Can Terraform deploy across multiple providers?

Yes, using provider plugins for seamless multi-cloud setups.

30. Handling duplicate resource errors in Terraform?

Use -ignore_duplicate to skip duplicates and continue deployment.

31. Components of Terraform Architecture?

CLI

Core engine

Providers

State management

32. What are Terraform's built-in functions?

Functions like format(), lookup(), and element() enable dynamic data handling in configurations.

32.Types of files in terraform

In Terraform, several types of files are used, each serving a specific purpose. Here's a quick overview:

1. .tf files

  - Purpose: Main configuration files.

  - Use: Define the infrastructure you want to create or manage, such as resources, providers, and variables.

  - Example:

```hcl
resource "aws_instance" "example" {
  ami           = "ami-0c55b159cbfafe1f0"
  instance_type = "t2.micro"
}
```

2. .tfvars files

- Purpose: Store variable values.
- Use: Provide input values for variables defined in .tf files, making configurations reusable and environment-specific.
- Example:

```hcl
instance_type = "t2.micro"
region        = "us-east-1"
```

3. .tfstate files

- Purpose: Keep track of the current state of the infrastructure.
- Use: Record the real-world infrastructure Terraform is managing, ensuring changes are applied correctly.
- Example: JSON file generated and updated automatically during terraform apply.

4. terraform.lock.hcl

- Purpose: Lock provider versions.
- Use: Ensure consistent provider versions across different runs or team members.
- Example:

```hcl
provider "aws" {
  version = "4.0.0"
}
```

5. .backup files

- Purpose: Backup of the previous .tfstate file.

- Use: Automatically created as a safety measure when .tfstate is updated.

6. Modules Directory (/modules/)

- Purpose: Reusable code blocks.

- Use: Group related .tf files for modular, reusable components like a VPC or EC2 setup.

7. Provider Plugins (.terraform directory)

- Purpose: Store provider binaries.

- Use: Needed to interact with specific cloud platforms or services like AWS or Azure.

8. Output Files (.out files)

- Purpose: Store plans.

- Use: Used by terraform plan to save a preview of the changes that terraform apply will make.

These files work together to make Terraform powerful and organized, supporting a seamless infrastructure management process!

33. Terraform Validation

What is Terraform Validation?

Terraform validation is a process that checks the syntax and configuration of your .tf files to ensure they are properly formatted and logically consistent before applying changes. It ensures that your Terraform code is free of errors and meets the required structure for Terraform to execute.

How It Works

When you run the terraform validate command:

1. Syntax Check: Verifies the structure of the .tf files to ensure there are no syntax errors (e.g., missing braces, typos).

2. Configuration Check: Validates that the configuration references correct resources, modules, and attributes.

Key Points

- It does NOT check the actual infrastructure or validate if the configurations can successfully create resources (e.g., missing permissions).

- It's a pre-apply step to catch errors early.

- It works only if the Terraform code is initialized (terraform init must be run first).

Example

Suppose you have a file main.tf with this configuration:

```
hcl
resource "aws_instance" "example" {
  ami           = "ami-0c55b159cbfafe1f0"
  instance_type = "t2.micro"
}
```

If there's a syntax issue, like a missing closing brace:

```
hcl
resource "aws_instance" "example" {
  ami           = "ami-0c55b159cbfafe1f0"
  instance_type = "t2.micro"
```

Running terraform validate will show an error:

```
Error: Missing closing brace
```

Once corrected, terraform validate will return:

```
Success! The configuration is valid.
```

Why Use Terraform Validation?

1. Catch Errors Early: Prevents running invalid configurations that could fail during terraform plan or apply.

2. Maintain Quality: Ensures your code is syntactically correct and logically consistent.

3. Best Practice: Regular validation helps avoid deployment delays caused by typos or misconfigurations.

Command

Run the following in the terminal:

bash

```
terraform validate
```

This is an essential step in a Terraform workflow, especially in CI/CD pipelines.

#Mention the provider and region in which you want to create the AWS resources in provider.tf file.

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

#Then create a variables file where you have to mention all the variables that is EC2 instance going to use such as ami-id , instance type , server port , ssh port in variables.tf file.

CODE EXAMPLE:

```
----------------
variable "instance_type" {
   description = "Instance type"
   type = string
   default = "t2.micro"
}
```

```
variable "ami_id" {
   description = "ami image"
   type = string
   default = "ami-0102332addane"
}
variable "server_port" {
   description = "for server port"
   type = number
   default = 80
}
variable "ssh_port" {
   description = "ssh port"
   type = number
   default = "22"
}
variable "availability_zone" {
   description = "Availability zone"
   default = "us-east-1"
}
```

#Then create a file where you can define the security group configurations in security_group.tf file.

```
resource "aws_security_group" "instance" {
   name = "terraform-sg"
   ingress {
      from_port = var.server_port
      to_port =var.server_port
      protocol ="tcp"
      cidr_blocks = ["0.0.0.0/0"]
   }
```

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

#Then create aws instance resource . Here mentions the variables and also in the user data write a script that will install httpd service and index.html on the EC2 instance in main.tf terraform file.

```
resource "aws_instance" "server" {
  ami = var.ami_id
  instance_type = var.instance_type
  vpc_security_group_ids = [aws_security_group.instance.id]
  availability_zone = var.availability_zone
  tags = {
    Name = "EC2-Server"
  }
  user_data = << -EOF
          #!/bin/bash
          sudo yum update -y
          sudo yum install httpd -y
          sudo systemctl start httpd
          systemctl enable httpd
          echo "Terraform is easy!!!">
  /var/www/html/index.html
          EOF
    user_data_replace_on_change = true
}
```

#Create an EBS volume . Then attach it to the EC2 instance in ebs_volume.tf file.

```
resource "aws_volume_attachment" "ebs_att" {
  device_name = "/dev/sdh"
  volume_id = aws_ebs_volume.ebs_vol.id
  instance_id = aws_instance.server.id
}

resource "aws_ebs_volume" "ebs_vol" {
  availability_zone = var.availability_zone
  size = 1
}
```

#Create a output file which will give the public IP of the EC2 instance as output in output.tf terraform file.

```
output "public_ip" {
  description = "The public IP address of the web serever"
  value = aws_instance.server.public_ip
}
```

IMPORTANT commands

Basic Operations

✓ terraform init - Initialize a new or existing Terraform configuration

✓ terraform validate - Validate the Terraform configuration files

✓ terraform fmt - Format configuration files to canonical style

✓ terraform version - Display the installed Terraform version

✓ terraform providers - List providers required by the current configuration

Workspace Management

✓ terraform workspace list - List all Terraform workspaces

✓ terraform workspace new <name> - Create a new workspace

✓ terraform workspace select <name> - Switch to a different workspace

✓ terraform workspace show - Display the current workspace

Planning

✓ terraform plan - Generate and show an execution plan

✓ terraform plan -out=<plan-name> - Save the execution plan to a file

Execution

✓ terraform apply - Apply changes to infrastructure

✓ terraform apply <plan-file> - Apply a saved execution plan

✓ terraform destroy - Destroy Terraform-managed infrastructure

✓ terraform destroy -target=<resource> - Destroy a specific resource

State Management

✓ terraform state list - List all resources in the state

✓ terraform state show <resource> - Show detailed information about a resource

✓ terraform state rm <resource> - Remove a resource from the state file

✓ terraform state mv <source> <destination> - Move a resource in the state file

Output and Variables

✓ terraform output - Show all output values

✓ terraform output <output-name> - Display a specific output value

✓ terraform variables - List all variables in the configuration

Debugging

✓ terraform show - Show state or plan details in a human-readable format

✓ terraform refresh - Update state to reflect real-world resources

✅ terraform graph - Generate a visual graph of resources

Providers and Plugins

✅ terraform providers - List required providers

✅ terraform login - Authenticate Terraform CLI with Terraform Cloud

Import and Export

✅ terraform import <resource> <ID> - Import an existing resource into the state

✅ terraform state pull - Fetch the current state file

✅ terraform state push - Overwrite the remote state with a local state file

Locks and Unlocks

✅ terraform force-unlock <lock-id> - Forcefully unlock the state file

Advanced Operations

✅ terraform plan -destroy - Preview what resources will be destroyed

✅ terraform apply -auto-approve - Apply changes without prompting for approval

✅ terraform taint <resource> - Mark a resource for recreation

✅ terraform untaint <resource> - Unmark a resource for recreation

✅ terraform console - Interactive console for evaluating expressions

💜 Bonus: Automation Tips 💜

✅ terraform init -backend-config="key=value" - Initialize with specific backend configuration

✅ terraform apply -var="key=value" - Pass variable values directly

✅ terraform apply -var-file="file.tfvars" - Pass a variable file during execution