Terraform Most asked Interview questions.

What is Terraform?
Terraform is an infrastructure-as-code (IaC) tool. Think of it as a way to write code that describes the infrastructure you want to build (like servers, networks, databases, etc.) in a cloud provider like Google Cloud, AWS, or Azure. Here's why it's so powerful:

- **Declarative:** You tell Terraform *what* you want, not *how* to do it. It figures out the steps to get there.
- **Repeatable:** You can run Terraform again and again, and it will ensure your infrastructure matches your code.
- **Version Control:** Store your Terraform code in version control (like Git) so you can track changes, collaborate, and easily roll back to previous versions.

Why Terraform?
Terraform addresses the challenges of managing infrastructure in a modern, cloud-centric world. Here's a breakdown of its key benefits:

1. **Infrastructure as Code (IaC):**
    - **Consistency:** Terraform allows you to define your infrastructure using code, ensuring that deployments are consistent and repeatable. No more manual configuration errors!
    - **Version Control:** Store your Terraform code in version control (like Git). This lets you track changes, collaborate with others, and easily roll back to previous versions if needed.
    - **Documentation:** Your Terraform code itself serves as documentation for your infrastructure, making it easier to understand and maintain.
2. **Automation:**
    - **Speed and Efficiency:** Terraform automates the process of creating, updating, and destroying infrastructure resources. This saves you time and effort, especially when dealing with complex deployments.
    - **Reduced Errors:** Automation minimizes the risk of human error during infrastructure management.
3. **Scalability:**
    - **Multi-Cloud Support:** Terraform works with major cloud providers (Google Cloud, AWS, Azure, etc.), allowing you to manage infrastructure across different platforms.
    - **Large-Scale Deployments:** Terraform can handle complex deployments with many resources, making it ideal for large-scale projects.
4. **Collaboration:**
    - **Teamwork:** Terraform facilitates collaboration among teams by providing a shared, code-based approach to infrastructure management.
5. **Planning and Preview:**
    - **Execution Plans:** Before making any changes, Terraform generates an execution plan that shows you exactly what will be created, updated, or destroyed. This allows you to review and approve changes before they are applied.

Benefits of Terraform?

Terraform is a powerful tool for managing infrastructure as code (IaC). Here's why it's so beneficial:

1. **Automation:**
    - **Speed and Efficiency:** Terraform automates the process of creating, updating, and destroying infrastructure resources. This saves you time and effort, especially when dealing with complex deployments.
    - **Reduced Errors:** Automation minimizes the risk of human error during infrastructure management.
2. **Consistency:**
    - **Repeatable Deployments:** Terraform ensures that your infrastructure is always in the desired state. You can run Terraform again and again, and it will always create or update your infrastructure to match your code.
    - **Version Control:** Store your Terraform code in version control (like Git). This lets you track changes, collaborate with others, and easily roll back to previous versions if needed.
3. **Collaboration:**
    - **Shared Code:** Terraform facilitates collaboration among teams by providing a shared, code-based approach to infrastructure management.
    - **Documentation:** Your Terraform code itself serves as documentation for your infrastructure, making it easier to understand and maintain.
4. **Planning and Preview:**
    - **Execution Plans:** Before making any changes, Terraform generates an execution plan that shows you exactly what will be created, updated, or destroyed. This allows you to review and approve changes before they are applied.

Terraform workflow ?

1. Init (Initialize the Working Directory)

- **Purpose:** The terraform init command initializes your Terraform working directory. It does the following:
    - **Downloads Providers:** Terraform downloads the necessary plugins (providers) for the resources you're defining in your code. In your case, since you're using the local provider, Terraform will download the necessary plugin for local file operations.
    - **Creates State File:** Terraform creates a state file (terraform.tfstate) to track the current state of your infrastructure. This file is crucial for Terraform to understand what's already deployed and what needs to be changed.
    - **Sets Up Backend:** If you're using a remote backend (like Google Cloud Storage or AWS S3) to store your state file, terraform init will configure the backend connection.

2. Plan (Preview Changes)

- **Purpose:** The terraform plan command analyzes your Terraform code and creates an execution plan. It shows you what changes Terraform will make to your infrastructure if you run terraform apply.
- **Output:** The terraform plan command will output a detailed plan, including:
    - **Resources to Create:** Lists the resources that will be created.
    - **Resources to Update:** Lists the resources that will be updated.
    - **Resources to Destroy:** Lists the resources that will be destroyed.

        - **Changes:** Shows the specific changes that will be made to each resource.

3. Apply (Apply Changes)

- **Purpose:** The terraform apply command executes the plan created by terraform plan. It makes the actual changes to your infrastructure.
- **Confirmation:** Before applying the changes, Terraform will ask you to confirm that you want to proceed.
- **Output:** The terraform apply command will output information about the changes that were made, including:
    - **Resources Created:** Lists the resources that were created.
    - **Resources Updated:** Lists the resources that were updated.
    - **Resources Destroyed:** Lists the resources that were destroyed.

4. Destroy (Clean Up Resources)

- **Purpose:** The terraform destroy command removes the resources that were created by Terraform.
- **Confirmation:** Terraform will ask you to confirm that you want to destroy the resources.
- **Output:** The terraform destroy command will output information about the resources that were destroyed.

Terraform Command Lines

Terraform CLI tricks

- **terraform -install-autocomplete** #Setup tab auto-completion, requires logging back in

Format and Validate Terraform code

- **terraform fmt** #format code per HCL canonical standard
- **terraform validate** #validate code for syntax
- **terraform validate -backend=false** #validate code skip backend validation

Initialize your Terraform working directory

- **terraform init** #initialize directory, pull down providers
- **terraform init -get-plugins=false** #initialize directory, do not download plugins
- **terraform init -verify-plugins=false** #initialize directory, do not verify plugins for Hashicorp signature

Plan, Deploy and Cleanup Infrastructure

- **terraform apply --auto-approve** #apply changes without being prompted to enter "yes"
- **terraform destroy --auto-approve** #destroy/cleanup deployment without being prompted for “yes”
- **terraform plan -out plan.out** #output the deployment plan to plan.out
- **terraform apply plan.out** #use the plan.out plan file to deploy infrastructure

- **terraform plan -destroy** #outputs a destroy plan
- **terraform apply -target=aws_instance.my_ec2** #only apply/deploy changes to the targeted resource
- **terraform apply -var my_region_variable=us-east-1** #pass a variable via command-line while applying a configuration
- **terraform apply -lock=true** #lock the state file so it can't be modified by any other Terraform apply or modification action(possible only where backend allows locking)
- **terraform apply refresh=false** # do not reconcile state file with real-world resources(helpful with large complex deployments for saving deployment time)
- **terraform apply --parallelism=5** #number of simultaneous resource operations
- **terraform refresh** #reconcile the state in Terraform state file with real-world resources
- **terraform providers** #get information about providers used in current configuration

Terraform Workspaces

- **terraform workspace new mynewworkspace** #create a new workspace
- **terraform workspace select default** #change to the selected workspace
- **terraform workspace list** #list out all workspaces

Terraform State Manipulation

- **terraform state show aws_instance.my_ec2** #show details stored in Terraform state for the resource
- **terraform state pull > terraform.tfstate** #download and output terraform state to a file
- **terraform state mv aws_iam_role.my_ssm_role module.custom_module** #move a resource tracked via state to different module
- **terraform state replace-provider hashicorp/aws registry.custom.com/aws** #replace an existing provider with another
- **terraform state list** #list out all the resources tracked via the current state file
- **terraform state rm aws_instance.myinstace** #unmanage a resource, delete it from Terraform state file

Terraform Import And Outputs

- **terraform import aws_instance.new_ec2_instance i-abcd1234** #import EC2 instance with id i-abcd1234 into the Terraform resource named "new_ec2_instance" of type "aws_instance"
- **terraform import 'aws_instance.new_ec2_instance[0]' i-abcd1234** #same as above, imports a real-world resource into an instance of Terraform resource
- **terraform output** #list all outputs as stated in code
- **terraform output instance_public_ip** # list out a specific declared output
- **terraform output -json** #list all outputs in JSON format

Terraform Miscelleneous commands

- **terraform version** #display Terraform binary version, also warns if version is old
- **terraform get -update=true** #download and update modules in the "root" module.

Terraform Console(Test out Terraform interpolations)

- **echo 'join(",",["foo","bar"])' | terraform console** #echo an expression into terraform console and see its expected result as output
- **echo '1 + 5' | terraform console** #Terraform console also has an interactive CLI just enter "terraform console"
- **echo "aws_instance.my_ec2.public_ip" | terraform console** #display the Public IP against the "my_ec2" Terraform resource as seen in the Terraform state file

Terraform Graph(Dependency Graphing)

- **terraform graph | dot -Tpng > graph.png** #produce a PNG diagrams showing relationship and dependencies between Terraform resource in your configuration/code

Terraform Taint/Untaint(mark/unmark resource for recreation -> delete and then recreate)

- **terraform taint aws_instance.my_ec2** #taints resource to be recreated on next apply
- **terraform untaint aws_instance.my_ec2** #Remove taint from a resource
- **terraform force-unlock LOCK_ID** #forcefully unlock a locked state file, LOCK_ID provided when locking the State file beforehand

Terraform Cloud

- **terraform login** #obtain and save API token for Terraform cloud
- **terraform logout** #Log out of Terraform Cloud, defaults to hostname app.terraform.io

Terraform provisioners types There are three types of provisioners in Terraform:

- Local-exec provisioners
- Remote-exec provisioners
- File provisioners

1. Local-Exec Provisioners

- **Purpose:** Run commands directly on the machine where Terraform is running. This is useful for tasks like:
  - Installing software
  - Configuring services
  - Running scripts

Syntax

```
provisioner "local-exec" {
  command = "echo 'Hello from local-exec!'"
}
```

Example:

```
resource "aws_instance" "example" {
  # ... (your EC2 instance configuration) ...

  provisioner "local-exec" {
    # Assuming you have SSH access to the instance
    command = "ssh ec2-user@${aws_instance.example.public_ip} 'sudo yum install -y httpd'"
  }
}
```

2. Remote-Exec Provisioners

- **Purpose:** Run commands on a remote machine, typically using SSH or WinRM. This is useful for tasks like:
  - Configuring remote servers
  - Running scripts on remote machines
- **Syntax**

```
provisioner "remote-exec" {
  connection {
    type        = "ssh"
    host        = "your-remote-host"
    user        = "your-username"
    private_key = file("path/to/your/private_key")
  }
  inline = [
    "echo 'Hello from remote-exec!'"
  ]
}
```

Example

```
resource "null_resource" "example" {
  provisioner "remote-exec" {
    connection {
      type        = "ssh"
      host        = "your-remote-host"
      user        = "your-username"
      private_key = file("path/to/your/private_key")
    }
    inline = [
      "sudo bash /path/to/your/script.sh"
    ]
  }
}
```

What is a null_resource?

- **Purpose:** The null_resource is a placeholder resource that doesn't create or manage any actual infrastructure. It's primarily used for:
    - **Running Provisioners:** You can attach provisioners (like local-exec, remote-exec, or file) to a null_resource to run commands or copy files after a resource is created or updated.
    - **Triggering Actions:** You can use a null_resource to trigger actions based on events or conditions.
    - **Custom Logic:** You can use a null_resource to implement custom logic or workflows within your Terraform code.

Why Use a null_resource?

- **Flexibility:** It allows you to perform actions that aren't directly supported by other Terraform resources.
- **Provisioning:** It's a convenient way to run provisioners after a resource is created or updated.
- **Workflows:** You can use it to create custom workflows and logic within your Terraform code.

```
resource "aws_instance" "example" {
  # ... (your EC2 instance configuration) ...
}

resource "null_resource" "run_script" {
  provisioner "remote-exec" {
    connection {
      type        = "ssh"
      host        = aws_instance.example.public_ip
      user        = "ec2-user"
      private_key = file("path/to/your/private_key")
    }
    inline = [
      "sudo bash /path/to/your/script.sh"
    ]
  }
  # Ensure the script runs after the EC2 instance is created
  depends_on = [aws_instance.example]
}
```

3. File Provisioners

- **Purpose:** Copy files to a remote machine. This is useful for tasks like:
    - Deploying application code
    - Copying configuration files
- **Syntax**

```
provisioner "file" {
  source      = "path/to/your/file"
  destination = "/path/on/remote/machine"
}
```

Example:

```
resource "null_resource" "example" {
  provisioner "file" {
    source      = "path/to/your/config.json"
    destination = "/etc/your-app/config.json"
  }
}
```

What is a Terraform provider?

Answer: A provider in Terraform is a plugin that allows Terraform to interact with APIs of cloud providers, SaaS providers, or other services. Examples include AWS, Azure, GCP, and Kubernetes.

Types of Terraform block ?

1. Provider Block

The provider block specifies the provider that Terraform will use to interact with your cloud or service infrastructure. It includes configuration settings for the provider, such as authentication credentials and region settings.

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

2. **2. Resource Block**

The resource block defines the infrastructure components you want to manage. Each resource block specifies a resource type and a name, along with its configuration parameters.

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

In this example, aws_instance is the resource type, and example is the resource name. The block contains parameters such as ami and instance_type.

3. Data Block

The data block allows you to fetch or reference data from providers. Unlike resource blocks, data blocks are used to retrieve information about existing infrastructure rather than managing or creating new resources.

```
data "aws_ami" "latest_amazon_linux" {
  owners = ["amazon"]
  most_recent = true
  filters = {
    name = "amzn2-ami-hvm-*-x86_64-gp2"
  }
}
```

4. Output Block
The output block defines outputs that Terraform will display after applying the configuration. Outputs are useful for displaying values like resource IDs or IP addresses, which can be used by other configurations or scripts.
Example:

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

5. Variable Block
The variable block defines input variables that allow you to parameterize your configuration. Variables help make configurations reusable and flexible by allowing values to be set externally.

```
variable "instance_type" {
  description = "Type of the instance"
  default     = "t2.micro"
}
```

6. Local Block

The locals block allows you to define local values that can be used throughout your configuration. These values are computed once and then used in other parts of the configuration.

```
locals {
  instance_name = "my-instance"
}
```

7. Module Block

The module block is used to include reusable configurations from other .tf files or directories. Modules allow you to encapsulate and reuse configurations, making it easier to manage complex infrastructure.

```
module "network" {
  source = "./network-module"
  vpc_name = "my-vpc"
}
```

8. Terraform Block

The terraform block configures settings related to Terraform itself, such as backend configuration, required providers, and required Terraform version.

```
terraform {
  required_version = ">= 1.0"

  backend "s3" {
    bucket         = "my-terraform-state"
    key            = "terraform.tfstate"
    region         = "us-east-1"
  }
}
```

Terraform state file ?

The Terraform state file (`terraform.tfstate`) is a critical component in Terraform's infrastructure management. It tracks the current state of your infrastructure as defined in your Terraform configurations. Here's a brief overview:

1. **State Tracking**: It records the real-world state of your infrastructure, mapping resources defined in your Terraform files to their current status in the cloud provider or other infrastructure platform.
2. **Metadata Storage**: It stores metadata about resources, which helps Terraform understand what is currently deployed and manage changes efficiently.
3. **Change Management**: Terraform uses the state file to plan and apply changes, ensuring that updates to your infrastructure are executed correctly and consistently.
4. **Remote State**: For teams or collaborative environments, the state file can be stored remotely (e.g., in an S3 bucket, Terraform Cloud, or other remote backends) to enable sharing and locking.

Managing and securing the state file is crucial since it contains sensitive information

Common Remote State Backends:

- **Amazon S3 with DynamoDB Locking**: Stores state in an S3 bucket and uses DynamoDB for locking and consistency.
- **Azure Blob Storage**: Stores state in Azure Blob Storage with optional state locking via Azure Table Storage.
- **Google Cloud Storage (GCS)**: Uses GCS for state storage, with optional locking through GCS object versioning.

- **Terraform Cloud/Enterprise**: Provides state management, locking, and versioning features integrated into Terraform's own platform.
- **Consul**: A distributed key-value store that can be used for storing Terraform state and provides locking.

```
terraform {
  backend "s3" {
    bucket         = "my-terraform-state-bucket"
    key            = "terraform/state.tfstate"
    region         = "us-west-2"
    dynamodb_table = "my-lock-table"  # Optional for state locking
    encrypt        = true             # Optional for encryption
  }
}
```

Resource Dependencies

- **depends_on Attribute:** Specify dependencies between resources using the depends_on attribute within resource blocks. This ensures that one resource is created before another.

  Destroying a Single Resource

- **terraform destroy -target:** Use the terraform destroy -target command followed by the resource type and name to destroy a specific resource.
  - Example: terraform destroy -target=aws_instance.my_instance

Modules – Local and Remote

Both local and distant sources can be used to load modules. The majority of version control systems, HTTP URLs, the Terraform Registry, and private module registries in Terraform Cloud or Terraform Enterprise are just a few of the external sources that Terraform supports.

What does a module do?

By building logical abstraction on top of a resource set, you may use modules in terraform. To put it another way, a module enables the grouping of resources for subsequent reuse—possibly repeatedly.

Assume we have a cloud-hosted virtual server with various functionalities. Which group of resources would best sum up that server? For instance:

1. A virtual machine built from a picture
2. A block device with a connected storage space that is a specific size.
3. A public static IP address that is assigned to the server’s virtual network interface
4. A set of firewall guidelines that should be attached to the server

5. Other factors, such as an extra network interface or block device.

Let’s now imagine that you will frequently need to establish this server with a certain set of resources. You don’t want to keep repeating the same setup code, do you? That’s where modules come in incredibly handy.

Why Modules?

- **Organise Configuration: Easier to navigate, understand, and update your configuration**
- **Encapsulate Configuration: Helps prevent unintended consequences**
- **Re-use Configuration: Share modules that you have written with your team or the general public**
- **Consistency: help to provide consistency in your configurations**

Examples:

```
resource "aws_instance" "appserver" {
    ami = var.ami
    instance_type = "t2.medium"
    tags = {
        name = "${var.app_region}-app-server"
    }
}

module "primaryapp" {
    source = "./modules/appstorage"
    app_region = "us-east-2"
    ami = "ami-0010d386b82bc06f0"
}
```

What is Terraform Workspace?

- **Purpose: Workspaces allow you to manage separate states for the same set of Terraform configuration files. This means you can use the same code to deploy different versions of your infrastructure to different environments (like development, staging, and production) without them interfering with each other.**
- **How it Works:**
  - **State Separation: Each workspace maintains its own state file, which is a record of the infrastructure managed by Terraform within that workspace.**
  - **Environment Isolation: This isolation prevents conflicts when you're working on different environments concurrently.**
  - **Code Reusability: You can use the same Terraform code for all environments, but each workspace will have its own state, ensuring that changes in one environment don't affect others.**

Example: Using Workspaces

Let's say you want to deploy your Terraform code to both a development and a production environment. You can use workspaces to manage these environments separately:

1. **Create Workspaces:**

```
terraform workspace new dev
terraform workspace new prod
```

2. **Select workspaces:**

```
terraform workspace select dev
```