Terraform – Fully Detailed Notes and Common error and Solutions

---

1. What is Terraform? (Deep Explanation)

Terraform is a **DevOps tool** used to **create, update, and delete infrastructure automatically using code**.

Infrastructure means:

- Servers (EC2, VM)
- Storage (S3, disks)
- Network (VPC, subnets)
- Load balancers
- Databases

Terraform code is called **Infrastructure as Code (IaC)**.

Why Terraform is Needed (Real Life)

Manual cloud work causes problems:

- Takes more time
- Human mistakes
- No tracking
- Hard to repeat

Terraform solves this by:

- Automation
- Same setup every time
- Easy changes
- Easy destroy

---

2. Infrastructure as Code (IaC) – Fully Explained

IaC means **your infrastructure is written like a program**.

Without IaC

- Click AWS console
- No history
- Difficult rollback

With IaC

- Write `.tf` files
- Store in Git
- Review changes
- Rollback easily

IaC Benefits

- Speed
- Accuracy
- Consistency
- Team collaboration

---

3. Terraform Architecture (Internals)

Terraform works in **4 steps**:

1. Read configuration files
2. Check current state
3. Create execution plan
4. Apply changes

Components

1️⃣ Terraform Core

- Main engine
- Reads `.tf` files
- Creates plan

2️⃣ Providers

- Connect Terraform to cloud
- Example: AWS, Azure, GCP

3️⃣ Resources

- Actual infrastructure

4️⃣ State File

- Keeps mapping between code & real infra

---

4. Terraform Installation (Detailed)

Linux / Windows / Mac

1. Download Terraform binary
2. Extract file
3. Move to PATH
4. Verify using:
   `terraform -version`

Terraform is **single binary tool** (no dependencies).

---

5. Terraform Configuration Files

Terraform reads **all `.tf` files automatically**.

Common Files

- `main.tf` → main logic
- `provider.tf` → cloud provider
- `variables.tf` → input values
- `outputs.tf` → output values

File names don't matter, **content matters**.

---

6. Terraform Provider (Very Detailed)

Provider tells Terraform **where to create infra**.

Example

AWS Provider needs:

- Region
- Credentials

Terraform downloads provider plugins during `init`.

One project can have **multiple providers**.

---

7. Terraform Resources (Deep Dive)

Resources are **building blocks**.

Each resource has:

- Type (aws_instance)
- Name (web)
- Arguments (AMI, instance type)

Terraform tracks resource using:
`resource_type.resource_name`

---

8. Terraform Variables (In-Depth)

Variables allow **dynamic values**.

Why Important

- Environment separation
- Reusability
- Clean code

Variable Types

- string
- number
- bool
- list
- map
- object

Variable Sources

- variables.tf
- terraform.tfvars
- CLI
- Environment variables

---

9. Terraform Output Values

Outputs show **important info after apply**.

Usage

- Show IPs
- Pass values to other tools
- Debug infra

Outputs are printed after `terraform apply`.

---

10. Terraform State File (Most Important Topic)

State file stores **real infrastructure details**.

Why Needed

- Know what exists
- Avoid duplication
- Manage changes

State Types

Local State

- Stored on local system
- Not safe for teams

Remote State

- S3
- Azure Blob
- Terraform Cloud

Remote state provides:

- Locking
- Team access
- Safety

---

11. Terraform Commands (Detailed Meaning)

terraform init

- Initialize directory
- Download providers
- Configure backend

terraform validate

- Syntax check

terraform plan

- Dry run
- Shows changes

terraform apply

- Executes plan
- Creates infra

terraform destroy

- Deletes infra

---

12. Terraform Lifecycle (How Infra Changes)

Terraform lifecycle:

1. Write code
2. Init
3. Plan
4. Apply
5. Update
6. Destroy

Terraform always compares:
Desired state vs Current state

---

13. Meta-Arguments (Advanced but Important)

depends_on

Force dependency

count

Create multiple resources

for_each

Loop with map or set

lifecycle

Control create/destroy behavior

---

14. Terraform Modules (Very Important for DevOps)

Modules = **Reusable Terraform code**.

Benefits

- DRY (Don’t Repeat Yourself)
- Clean structure
- Easy maintenance

Types

- Root module
- Child module

Used heavily in **real projects**.

---

15. Terraform Workspaces (Environment Management)

Workspaces allow **multiple environments**:

- dev
- test
- prod

Each workspace has **separate state file**.

---

16. Terraform Backend (State Storage)

Backend defines **where state is stored**.

Popular Backends

- S3 + DynamoDB
- Azure Blob
- Terraform Cloud

Backend is configured only once.

---

17. Provisioners (Last Option Tool)

Provisioners run scripts **after resource creation**.

Types

- local-exec
- remote-exec
- file

⚠️ Not recommended unless necessary.

---

18. Terraform Best Practices (Industry Level)

- Use remote state
- Use modules
- Use variables
- Never commit state
- Use Git
- Small modules
- Proper naming

---

19. Terraform vs Other Tools (Clear Difference)

Terraform vs Ansible

- Terraform → Infrastructure
- Ansible → Configuration

Terraform vs CloudFormation

- Terraform → Multi-cloud
- CloudFormation → AWS only

---

20. Terraform Real-World DevOps Usage

Used in:

- CI/CD pipelines
- Auto scaling infra
- Cloud migration
- Disaster recovery

Companies use Terraform **daily**.

---

21. Interview Important Points

- Terraform is declarative
- State file is critical
- Modules are reusable
- Plan before apply

---

22. One-Line Final Summary

Terraform lets you safely and automatically manage cloud infrastructure using simple, readable code.

✅ Common Terraform Errors & Their Solutions

(Real-world + Interview-important)

Common error and Solutions

1️⃣ `terraform: command not found`

❌ Why it happens

- Terraform not installed
- PATH not configured

✅ Solution

- Install Terraform properly
- Add Terraform binary to PATH
- Check:

```
terraform -version
```

2️⃣ `Provider configuration not present`

❌ Why it happens

- Provider block deleted
- Using resource without provider

✅ Solution

- Add provider block again
- Run:

```
terraform init
```

3️⃣ Failed to load backend

❌ Why it happens

- S3 bucket not created
- Wrong region or bucket name

✅ Solution

- Create S3 bucket first
- Verify backend config
- Re-run:

```
terraform init
```

4️⃣ No valid credential sources found

❌ Why it happens

- AWS credentials not set
- Wrong access key

✅ Solution

```
aws configure
```

or use environment variables:

```
AWS_ACCESS_KEY_ID
AWS_SECRET_ACCESS_KEY
```

5️⃣ InvalidParameter: The image id does not exist

❌ Why it happens

- Wrong AMI ID
- AMI not available in region

✅ Solution

- Check correct AMI for region
- Update AMI ID

6️⃣ `Reference to undeclared input variable`

❌ Why it happens

- Variable used but not declared

✅ Solution

- Declare variable in `variables.tf`
- Check spelling

7️⃣ `Unsupported argument`

❌ Why it happens

- Wrong argument name
- Old provider version

✅ Solution

- Check official Terraform docs

- Upgrade provider version

8️⃣ `Cycle detected`

❌ Why it happens

- Resource A depends on B
- B also depends on A

✅ Solution

- Remove circular dependency
- Use `depends_on` carefully

9️⃣ `Resource already exists`

❌ Why it happens

- Resource created manually
- Terraform doesn't know about it

✅ Solution

```
terraform import
```

or delete resource manually

🔟 `State file locked`

❌ Why it happens

- Previous Terraform run crashed
- Another user using state

✅ Solution

- Wait for unlock
  OR (carefully):

```
terraform force-unlock LOCK_ID
```

1️⃣1️⃣ Invalid count argument

❌ Why it happens

- `count` uses dynamic value

✅ Solution

- Use static value
- Or replace with `for_each`

1️⃣2️⃣ Timeout while waiting for resource

❌ Why it happens

- Cloud API slow
- Resource creation taking time

✅ Solution

- Increase timeout
- Check cloud console

1️⃣3️⃣ terraform apply stuck / hanging

❌ Why it happens

- Network issue
- API throttling

✅ Solution

- Check internet
- Retry apply

1️⃣4️⃣ Invalid function argument

❌ Why it happens

- Wrong data type passed

✅ Solution

- Convert data types:

```
tolist()
tomap()
```

1️⃣5️⃣ Missing required argument

❌ Why it happens

- Mandatory field not added

✅ Solution

- Check resource documentation
- Add missing argument