TERRAFORM INTERVIEW PREP

1. NULL\_RESOURCE - The null\_resource allows you to:

* Run arbitrary **provisioners** (like local-exec or remote-exec) during apply.
* Trigger actions **based on changes** to input values (via triggers).
* Act as a **placeholder** or **dummy resource** when you need to run custom scripts or commands without actually creating cloud resources.

Syntax example

=================

resource "null\_resource" "example" {

provisioner "local-exec" {

command = "echo 'Hello, World!'"

}

triggers = {

always\_run = "${timestamp()}"

}

}

**Caution / Limitations**

* **Not idempotent by default**: If used with timestamp() or changing values, it can rerun too often.
* **Not for production lifecycle management**: Better to use proper resources or external tools if you need reliable scripting.

**When to Use It**

Use null\_resource when:

* You need a script to run conditionally within Terraform.
* You're bridging a gap Terraform can't handle natively.
* You’re orchestrating steps Terraform doesn’t support directly.
* Let's walk through a **practical example** of using a null\_resource as an **integration hook** to run a script after a cloud resource (like an AWS EC2 instance) is created.

provider "aws" {

region = "us-east-1"

}

resource "aws\_instance" "example" {

ami = "ami-0c55b159cbfafe1f0" # Use a valid AMI for your region

instance\_type = "t2.micro"

tags = {

Name = "Terraform-Example"

}

}

resource "null\_resource" "after\_instance\_creation" {

# This ensures the null\_resource depends on the EC2 instance

depends\_on = [aws\_instance.example]

# Run a script after instance is created

provisioner "local-exec" {

command = <<EOT

echo "Instance ID: ${aws\_instance.example.id}" >> instance\_info.txt

echo "Public IP: ${aws\_instance.example.public\_ip}" >> instance\_info.txt

curl -X POST -H "Content-Type: application/json" -d '{"instance\_id": "${aws\_instance.example.id}"}' https://your-webhook-endpoint.com/hook

EOT

}

}

1. In Terraform**, locals.tf, variables.tf, and terraform.tfvars** all play distinct roles in managing configuration values. Here's a clear breakdown of how each is used and how they differ in the context of handling variables:

✅ variables.tf – **Declares Input Variables**

* **Purpose**: Used to declare input variables that can be passed into your Terraform configuration.
* **What it contains**: Variable **definitions**, including type, default values (optional), and descriptions.

variable "region" {

type = string

description = "AWS region to deploy resources"

default = "us-west-2"

}

* **Use case**: You want to make your configuration reusable and flexible by allowing users to specify values at runtime.

✅ terraform.tfvars – **Supplies Values to Variables**

* **Purpose**: Used to provide **values** for the input variables declared in variables.tf.
* **What it contains**: Just variable names and their values, no type or descriptions.

region = "us-east-1"

* **Use case**: You want to separate config data (like environment-specific values) from your Terraform code

✅ locals.tf – **Defines Local Values (Immutable Constants)**

* **Purpose**: Used to define **constants or computed values** within the configuration.
* **What it contains**: Local variables that are derived from other values (e.g., input variables, resource attributes).

locals {

full\_region\_name = "${var.environment}-${var.region}"

}

* **Use case**: Simplify expressions, avoid repetition, or make complex values easier to manage.
* **Important**: Locals **cannot** be passed in from the outside like input variables. They’re purely internal.

Let’s walk through a **practical scenario** that shows how you can use variables.tf, terraform.tfvars, and locals.tf in a real Terraform project—especially to **separate environment-specific configuration data from your core Terraform code**.

👷 Scenario: Deploying Infrastructure to Multiple Environments (Dev & Prod)

Imagine you're using Terraform to provision AWS infrastructure. You want to deploy **the same infrastructure** (e.g., a VPC, EC2, etc.) to both **development** and **production** environments — but with **different settings** like instance type, region, or tags.

**📁 File Structure**

project/

├── main.tf

├── variables.tf

├── locals.tf

├── terraform.tfvars # For default environment (say, dev)

├── dev.tfvars # Dev environment-specific values

├── prod.tfvars # Prod environment-specific values

**🔹 1. variables.tf: Declare Input Variables**

variable "environment" {

type = string

description = "The environment to deploy to (dev, prod)"

}

variable "region" {

type = string

description = "AWS Region"

}

variable "instance\_type" {

type = string

description = "EC2 instance type"

}

**🔹 2. terraform.tfvars, dev.tfvars, prod.tfvars: Provide Environment-Specific Values**

These files provide actual values for different environments.

**🟢 dev.tfvars**

environment = "dev"

region = "us-west-1"

instance\_type = "t2.micro"

**🔴 prod.tfvars**

environment = "prod"

region = "us-east-1"

instance\_type = "m5.large"

You can apply them using:

terraform apply -var-file="dev.tfvars"

# or

terraform apply -var-file="prod.tfvars"

**🔹 3. locals.tf: Define Computed or Helper Values**

locals {

name\_prefix = "${var.environment}-app"

common\_tags = {

Environment = var.environment

ManagedBy = "Terraform"

}

}

1. **What does terraform init do?**

Terraform init is the **first command** you run when starting a new Terraform project. It **initializes** your Terraform working directory so it can download providers and prepare the backend.

**🔧 Internally, terraform init does the following:**

|  |  |
| --- | --- |
| **1️⃣** | **Downloads provider plugins (e.g., AWS, Azure, Google) based on your configuration (provider "aws" { ... })** |
| **2️⃣** | **Initializes the backend (e.g., local, S3) used to store Terraform state** |
| **3️⃣** | **Prepares the working directory for use with Terraform** |
| **4️⃣** | **Installs modules if any are used (module "xyz" { source = ... }**   |  | | --- | | **5️⃣ Checks Terraform version constraints (if specified in .terraform-version or required\_version)**  📁 What files/folders appear after terraform init?  After you run terraform init, you will see a new directory called:  .terraform/  Here’s what your folder might look like:  project/  ├── main.tf  ├── variables.tf  ├── terraform.tfvars  ├── locals.tf  ├── .terraform/ 👈 Created by terraform init  │ ├── providers/ 👈 Downloaded provider binaries  │ └── modules/ 👈 (if using remote modules)  ├── .terraform.lock.hcl 👈 Lock file for provider versions  📌 .terraform/ Directory (Don’t Commit)   * Holds provider binaries like terraform-provider-aws. * Contains internal metadata and plugin info. * Should not be committed to version control (.gitignore it).   📌 .terraform.lock.hcl (Commit This)   * This is the provider lock file. * It records the exact versions of providers used so that all users of the code use the same ones. * Helps with reproducibility and security.   You should commit this file to Git.   1. **How terraform workspaces are beneficial?**   Terraform workspaces can be useful when you want to manage multiple instances of the same infrastructure using the same codebase — without duplicating your Terraform configuration files.  **✅ What Are Terraform Workspaces?**   * **Workspaces** allow you to maintain **multiple state files** in the same Terraform project. * Each workspace has its **own separate state**. * By default, Terraform starts with the **default** workspace.   **🎯 When Are Workspaces Beneficial?**  Workspaces are useful in scenarios like:   * Deploying the **same infrastructure** to multiple **environments** (e.g., dev, staging, prod) using **one codebase**. * Running **parallel copies** of infrastructure that are similar but need **isolated state management**. * Managing **feature branches** or **testing environments**.   ⚠️ However, workspaces **do not fully isolate configuration** — just the **state files**. You still need to use variables or conditionals to customize behavior per environment.   1. **What does terraform refresh command do ?**   **✅ What Does terraform refresh Do?**  The terraform refresh command **updates the Terraform state file** to match the **real infrastructure** as it exists in the provider (e.g., AWS, Azure, GCP) **without changing the configuration**.  🧠 Think of it as: “Sync the Terraform state with the current reality in the cloud.”  **🔍 In More Detail:**   * Terraform **queries the actual infrastructure** (e.g., EC2 instances in AWS). * It **compares** that to what's recorded in the current **state file** (terraform.tfstate). * It **updates the state file** to reflect the current state of the real resources.   **❗Important:**   * terraform refresh does **not** change infrastructure. * It only affects the **local state file**. * It helps Terraform “catch up” if something was changed **outside of Terraform** (e.g., someone changed an EC2 tag manually in AWS Console). | | 1. **What happens when you change a resource's name or id in Terraform?**   Changing a resource block's name (label) or key attributes like id, name, etc., can cause resource recreation:   * Terraform will treat it as a new resource and will destroy the old one and create a new one, which can be disruptive.   # Before  resource "aws\_s3\_bucket" "my\_bucket" {  bucket = "my-old-bucket"  }  # After  resource "aws\_s3\_bucket" "my\_bucket" {  bucket = "my-new-bucket"  } | |

1. **What is the purpose of terraform taint, and when should you use it?**

Terraform taint **manually marks a resource for recreation** on the next apply. This is useful when a resource is **corrupted or misbehaving** but its configuration hasn't changed.

terraform taint aws\_instance.web

terraform apply

Terraform will destroy and recreate aws\_instance.web, even if the config is unchanged.

⚠️ Note: As of Terraform v0.15+, terraform taint is deprecated in favor of terraform apply -replace="resource.name".

1. **What’s the difference between count and for\_each, and when should you use one over the other?**

| Feature | count | for\_each |
| --- | --- | --- |
| Input type | Integer (number) | Map or Set of strings |
| Index access | count.index | each.key / each.value |
| Resource keys | Numeric (e.g., [0], [1]) | Named (e.g., ["dev"], ["prod"]) |
| Best for | Simple repetition | Named/Keyed collections (maps/sets) |

**✅** Use count for:

count = 3

✅ Use for\_each when working with maps or sets:

for\_each = toset(["dev", "prod", "test"])

1. **What are the risks of using terraform import, and how do you use it properly?**

Terraform import brings existing infrastructure into Terraform’s state management without writing the configuration.

Steps:

1. Write the resource block (must match the imported resource).
2. Import

terraform import aws\_instance.example i-1234567890abcdef0

**Risks:**

* Only imports the **state**, not the **configuration**.
* You must **manually write matching Terraform code**, or else terraform plan will destroy it.
* Can cause **drift** or **accidental deletion** if mismatched.

1. **How can you prevent Terraform from recreating resources unnecessarily during minor config changes?**

* lifecycle block with ignore\_changes:

resource "aws\_instance" "example" {

...

lifecycle {

ignore\_changes = [tags["Environment"]]

}

}

* Use prevent\_destroy to block accidental deletions:

lifecycle {

prevent\_destroy = true

}

1. **What is the Terraform lifecycle Block?**

The lifecycle block is used within a resource block to customize Terraform’s default behavior for:

* **create\_before\_destroy** – create a new resource before destroying the old one
* **prevent\_destroy** – protect critical resources from accidental deletion
* **ignore\_changes** – ignore certain attribute changes during plan/apply

**🚀 create\_before\_destroy**

**🔹 Use Case:**

* You’re managing an **S3 bucket** or **Load Balancer**, and changing a key attribute (like name) would cause it to be destroyed and recreated. But you don’t want **downtime**.

**✅ Solution:**

* Use create\_before\_destroy to ensure the new resource is created **first**, then the old one is destroyed.

1. **Looping Techniques in Terraform**

* **Basic Loop Over a List (Set)**

variable "instance\_names" {

type = list(string)

default = ["web", "app", "db"]

}

resource "aws\_instance" "ec2" {

**for\_each = toset(var.instance\_names)**

ami = "ami-0c55b159cbfafe1f0"

instance\_type = "t2.micro"

tags = {

Name = each.key

}

}

* **Loop Over a Map - Use Case: Create security group rules with different ports and protocols**

variable "ingress\_rules" {

type = map(object({

port = number

protocol = string

}))

default = {

ssh = {

port = 22

protocol = "tcp"

}

http = {

port = 80

protocol = "tcp"

}

}

}

resource "aws\_security\_group\_rule" "ingress" {

for\_each = var.ingress\_rules

type = "ingress"

from\_port = each.value.port

to\_port = each.value.port

protocol = each.value.protocol

cidr\_blocks = ["0.0.0.0/0"]

security\_group\_id = aws\_security\_group.main.id

}

* **Nested Map Looping -> Use Case: Tagging multiple resources with nested metadata**

variable "resource\_tags" {

type = map(map(string))

default = {

"web" = {

Owner = "Dev"

Env = "dev"

Project = "Website"

}

"db" = {

Owner = "DBA"

Env = "prod"

Project = "Database"

}

}

}

output "formatted\_tags" {

value = {

**for resource, tags in var.resource\_tags** :

resource => {

**for tag\_key, tag\_value in tags :**

tag\_key => upper(tag\_value)

}

}

}

Explanation :-

**So in this line:**

for resource, tags in var.resource\_tags :

The loop will iterate twice:

**First iteration:**

* + resource = "web"
  + tags = { Owner = "Dev", Env = "dev", Project = "Website" }

**Second iteration:**

* + resource = "db"
  + tags = { Owner = "DBA", Env = "prod", Project = "Database" }

**Then for each tags map, this inner loop:**

for tag\_key, tag\_value in tags :

tag\_key => upper(tag\_value)

* **Conditional Logic Inside Loops -> Use Case: Only create EC2 instances for enabled = true**

variable "instances" {

type = map(object({

enabled = bool

type = string

}))

default = {

"web" = {

enabled = true

type = "t2.micro"

}

"dev" = {

enabled = false

type = "t2.nano"

}

}

}

resource "aws\_instance" "selected" {

for\_each = {

for name, cfg in var.instances :

name => cfg if cfg.enabled

}

instance\_type = each.value.type

ami = "ami-0c55b159cbfafe1f0"

tags = {

Name = each.key

}

}

Only the "web" instance will be created.

* **Flattening a Nested List 🡪 Use Case: Generate a flat list of CIDR blocks from multiple subnets**

variable "vpc\_subnets" {

type = list(object({

name = string

cidr\_blocks = list(string)

}))

default = [

{

name = "public"

cidr\_blocks = ["10.0.1.0/24", "10.0.2.0/24"]

},

{

name = "private"

cidr\_blocks = ["10.0.3.0/24"]

}

]

}

output "all\_cidr\_blocks" {

value = flatten([

for subnet in var.vpc\_subnets : subnet.cidr\_blocks

])

}

**A black rectangle with white dots

AI-generated content may be incorrect.**

1. **Terraform: Printing Maps and Nested Map Values**

* Printing a Simple Map

variable "simple\_map" {

default = {

"name" = "app-server"

"env" = "prod"

"owner" = "devops"

}

}

output "all\_keys" {

value = keys(var.simple\_map)

}

output "all\_values" {

value = values(var.simple\_map)

}

# Direct Access:

output "env" {

value = var.simple\_map["env"]

}

**A black background with green dots

AI-generated content may be incorrect.**

* **Nested Maps with examples**

variable "nested\_map" {

default = {

"web" = {

"port" = 80

"ssl" = false

}

"db" = {

"port" = 3306

"ssl" = true

}

}

}

#Access nested value (e.g., DB port):

output "db\_port" {

value = var.nested\_map["db"]["port"]

}

#Using lookup() for nested:

output "web\_ssl\_status" {

value = lookup(lookup(var.nested\_map, "web", {}), "ssl", false)

}



Explanation :=

Inner lookup:

**lookup(var.nested\_map, "web", {})**

* Looks up the key "web" in var.nested\_map
* If "web" exists, it returns:

{

"port" = 80

"ssl" = false

}

* If "web" does **not** exist, it returns {} (an empty map)

Outer lookup:

**lookup(<result\_of\_inner\_lookup>, "ssl", false)**

* **Looks up the key "ssl" in the result of step 1**
* **If "ssl" exists, it returns its value (false in this case)**
* **If not, it returns the default value false**
* **Iterate Over Nested Map and Format**

output "formatted\_ports" {

value = {

for svc, conf in var.nested\_map :

svc => "Port: ${conf["port"]}, SSL: ${conf["ssl"]}"

}

}

A black rectangle with white text

AI-generated content may be incorrect.

1. **What is the difference between for and for\_each statement in terraform?**

**🆚 for vs for\_each in Terraform**

| **Feature** | **for Expression** | **for\_each Argument** |
| --- | --- | --- |
| **Used In** | **Expressions (locals, outputs, variables)** | **Resource & module blocks** |
| **Returns** | **List or Map** | **Creates multiple instances of a resource** |
| **Purpose** | **Transform data** | **Create multiple resources from a collection** |
| **Context** | **Inside locals, output, or any value** | **Inside resource or module blocks** |

**🔍 Side-by-Side Comparison**

**❓When to Use Which?**

| **Use Case** | **Use for** | **Use for\_each** |
| --- | --- | --- |
| **Transform or filter data** | **✅ Yes** | **❌ No** |
| **Generate a list/map** | **✅ Yes** | **❌ No** |
| **Create multiple resources** | **❌ No** | **✅ Yes** |
| **Loop inside locals or outputs** | **✅ Yes** | **❌ No** |
| **Loop in a resource/module block** | **❌ No** | **✅ Yes** |

1. **How to Append a New Mapped Value to an Existing Map**

**Using merge() Function**

# Original map

variable "base\_tags" {

default = {

"Environment" = "dev"

"Owner" = "DevOps"

}

}

# New key-value pair to append

locals {

extra\_tag = {

"Project" = "TerraformMigration"

}

# Merged map (base + new)

combined\_tags = merge(var.base\_tags, local.extra\_tag)

}

output "final\_tags" {

value = local.combined\_tags

}

A black screen with white text

AI-generated content may be incorrect.

1. Difference between Flatten() and Merge()

**🆚 flatten() vs merge() in Terraform**

| **Function** | **Input Type(s)** | **Output Type** | **Purpose** |
| --- | --- | --- | --- |
| flatten() | List of lists | Flat list | Flattens nested lists into a single list |
| merge() | Multiple maps | Single map | Combines multiple maps into one map |

**🔍 Side-by-Side Summary**

| **Feature** | **flatten()** | **merge()** |
| --- | --- | --- |
| Input | [[...], [...]] | {...}, {...}, ... |
| Output | [...] (flat list) | {...} (single merged map) |
| Handles | Lists of lists | Multiple maps |
| **Key behavior** | **N/A** | **Overwrites duplicate keys** |
| Use in for? | Yes, for nested loops | No, but can be used after |

1. **Use Cases with Complex Conditional Logic**

* **Enable Resource Only in Specific Environments -> You want to create a resource only in non-prod environments.**

variable "environment" {

default = "dev"

}

resource "aws\_s3\_bucket" "log\_bucket" {

**count = var.environment != "prod" ? 1 : 0**

bucket = "logs-${var.environment}"

acl = "private"

}

* Conditional Map Merging Based on Input

variable "add\_cost\_tags" {

type = bool

default = true

}

locals {

base\_tags = {

Environment = "dev"

Owner = "DevOps"

}

cost\_tags = var.add\_cost\_tags ? {

CostCenter = "IT-001"

Billing = "enabled"

} : {}

final\_tags = merge(local.base\_tags, local.cost\_tags)

}

output "final\_tags" {

value = local.final\_tags

If add\_cost\_tags = true, final\_tags becomes:

A black rectangle with white dots

AI-generated content may be incorrect.

If add\_cost\_tags = false, final\_tags becomes:

A black rectangle with white dots

AI-generated content may be incorrect.

* Only deploy resources that are marked as enabled.

variable "services" {

default = {

"app" = { enabled = true, port = 8080 }

"db" = { enabled = false, port = 3306 }

"auth" = { enabled = true, port = 9090 }

}

}

resource "null\_resource" "service" {

for\_each = {

for k, v in var.services :

k => v if v.enabled == true

}

triggers = {

name = each.key

port = each.value.port

}

}

* Multi-level Nested Conditions - > Enable logging **only in prod** and **if audit mode is enabled**.

variable "environment" {

default = "prod"

}

variable "audit\_mode" {

default = true

}

locals {

logging\_enabled = (

var.environment == "prod" && var.audit\_mode == true

) ? true : false

}

output "logging\_enabled" {

value = local.logging\_enabled

}

* Complex Dynamic Block Condition -> Only attach a secondary network interface if a condition is met.

variable "enable\_secondary\_eni" {

default = false

}

resource "aws\_instance" "example" {

ami = "ami-123456"

instance\_type = "t2.micro"

dynamic "network\_interface" {

for\_each = var.enable\_secondary\_eni ? [1] : []

content {

device\_index = 1

network\_interface\_id = aws\_network\_interface.secondary.id

}

}

}

1. Terraform Complex Functions — With Use Cases

* zipmap() — Combine Two Lists into a Map

locals {

names = ["dev", "prod", "qa"]

values = ["10.0.1.0/24", "10.0.2.0/24", "10.0.3.0/24"]

env\_map = zipmap(local.names, local.values)

}

A black rectangle with white dots

AI-generated content may be incorrect.

* contains() — Check if Value Exists in List

variable "region" {

default = "us-west-2"

}

locals {

allowed\_regions = ["us-east-1", "us-west-2"]

is\_allowed = contains(local.allowed\_regions, var.region)

}

* coalesce() / coalescelist() — First Non-Null Value

output "env" {

value = coalesce("", null, "dev", "prod")

}

✅ Output: "dev"

output "list" {

value = coalescelist([], ["fallback"], ["default"])

}

✅ Output: ["fallback"]