CAREER BYTE CODE
REALTIME PROJECTS PLATFORM

91 COUNTRIES 241k Learners
+32 471 40 89 08
CAREERBYTECODE.SUBSTACK.COM

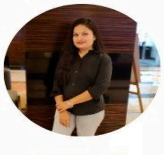
RealTime Handson

careerbytecode

Terraform

REALTIME SCENARIO BASED QUESTIONS AND ANSWER

PART 1



Sonali

techopsbysonali

1️⃣Scenario: Troubleshooting a Failing Terraform Apply

Question:
Your team has written a Terraform configuration to deploy an AWS EC2 instance. However, when running `terraform apply`, you receive the following error:

```
Error: UnauthorizedOperation: You are not authorized to perform this operation
```

How would you troubleshoot and resolve this issue?

Answer:
This error indicates that the IAM role or user running Terraform does not have the necessary permissions. To troubleshoot and fix:

1. **Check AWS Credentials:** Run `aws sts get-caller-identity` to verify the current IAM user/role.
2. **Review IAM Policies:** Ensure the user or role has permissions to create EC2 instances (`ec2:RunInstances`).
3. **Check Active Session:** If using temporary credentials, confirm they haven't expired.
4. **Verify Execution Role:** If running Terraform in an automation tool (e.g., GitHub Actions, Jenkins), ensure the correct role is assumed.
5. **Use `terraform plan` Debugging:** Run `terraform plan -out=tfplan` to identify specific resource failures before applying.

2️⃣Scenario: Managing State File Conflicts in a Team

Question:
Your team is collaborating on Terraform, but when multiple engineers try to apply changes, you frequently see errors related to state file locking. How would you prevent this issue?

Answer:
This issue occurs when multiple users attempt to modify the Terraform state simultaneously. Solutions include:

1. **Enable Remote State Locking:** Use Terraform Cloud, AWS S3 with DynamoDB, or GCP Cloud Storage with state locking.
2. For AWS S3, configure a DynamoDB table for locking:

```
backend "s3" {
  bucket         = "terraform-state-bucket"
  key            = "dev/terraform.tfstate"
  region         = "us-east-1"
  dynamodb_table = "terraform-lock"
}
```

○

3. **Use `terraform plan` Before Apply:** Always review planned changes before applying.
4. **Adopt a CI/CD Pipeline:** Automate Terraform deployments using a central pipeline instead of local execution.
5. **Restrict Manual Terraform Runs:** Ensure only specific users or roles have permission to apply Terraform changes.

3️⃣Scenario: Rollback After a Failed Terraform Deployment

Question:
You deployed infrastructure using Terraform, but a new change caused downtime. How do you roll back to a previous working state?

Answer:
To roll back after a failed deployment:

Use Version Control: Check out the last working version of Terraform code and reapply:

```
git checkout <previous_commit>
terraform apply
```

Restore from Remote State: If using a remote state backend, retrieve the last successful state and reapply:

```
terraform state pull > backup.tfstate
terraform apply -state=backup.tfstate
```

Manually Fix Configuration: If necessary, manually edit the configuration to fix the issue and reapply changes.

Terraform Destroy (Last Resort): If a rollback isn't feasible, destroy problematic resources and redeploy from scratch:

```
terraform destroy
terraform apply
```

4️⃣Scenario: Handling Sensitive Information in Terraform

Question:
You need to deploy an RDS database using Terraform and must store database credentials securely. How would you manage this?

Answer:
Sensitive information like database passwords should never be hardcoded in Terraform. Instead:

Use Terraform Variables: Store credentials as environment variables or use input variables:

```
variable "db_password" {
  type      = string
  sensitive = true
}
```

Use AWS Secrets Manager / Vault: Retrieve secrets dynamically instead of storing them in state files.

Example using AWS Secrets Manager:

```
data "aws_secretsmanager_secret_version" "db_creds" {
  secret_id = "my-db-secret"
}
```

Prevent State Exposure: Use the `sensitive = true` flag to hide values in Terraform output.

Use Remote State Encryption: Ensure that Terraform's state file is encrypted when stored remotely.

5️⃣Scenario: Terraform Resource Drift Detection

Question:
You suspect someone has manually modified resources outside of Terraform. How can you detect and correct this drift?

Answer:

Run `terraform plan`: Compare the current state with the desired configuration:

```
terraform plan
```

Use `terraform refresh`: Update the state file to reflect the real-world state:

```
terraform refresh
```

Implement Continuous Drift Detection: Use Terraform Cloud or GitHub Actions to periodically check for drift.

Manually Import Changes (if needed): If resources were created manually, import them into Terraform:

```
terraform import aws_instance.example i-12345678
```

Apply Terraform to Reconcile Drift: If resources are out of sync, apply the correct configuration:

```
terraform apply
```

6️⃣Scenario: Handling Module Versioning in Terraform

Question:
Your team uses Terraform modules stored in a private Git repository. A recent update to a module introduced breaking changes. How do you ensure that future Terraform applies do not break existing infrastructure?

Answer:
To handle module versioning and prevent breaking changes:

1. **Use Module Versioning with Tags/Branches:**
   - Instead of always using the latest module version, reference a specific tag in the Terraform module source.

```
module "networking" {
  source  = "git::https://github.com/myorg/networking-module.git?ref=v1.2.0"
}
```

2. **Test Changes in a Separate Environment:** Deploy updates in a staging environment before production.
3. **Use a Versioning Strategy:** Follow semantic versioning (`v1.0.0`, `v1.1.0`, `v2.0.0`) to track breaking changes.

Lock Module Versions in Terraform Registry:

```
module "networking" {
  source  = "myorg/networking/aws"
  version = "~> 1.2.0"
}
```

4. **Use `terraform plan` Before Apply:** Always validate the impact of module updates before applying.

7️⃣Scenario: Terraform Apply is Stuck Due to Pending Resource Deletion

Question:
You attempted to delete an S3 bucket using Terraform, but the operation is stuck because the bucket is not empty. How would you resolve this?

Answer:
S3 buckets cannot be deleted if they contain objects. To resolve this:

1. **Enable Force Delete in Terraform:**
   - Modify the Terraform configuration to delete objects before deleting the bucket.

```
resource "aws_s3_bucket" "example" {
  bucket = "my-bucket"
  force_destroy = true  # Enables automatic deletion of objects
}
```

2. **Manually Empty the Bucket:**

Use AWS CLI:

```
aws s3 rm s3://my-bucket --recursive
```

   - Then rerun `terraform apply`.

3. **Check IAM Permissions:** Ensure the Terraform execution role has `s3:DeleteObject` permission.

Retry the Terraform Apply:

```
terraform apply -auto-approve
```

8️⃣Scenario: Managing Cross-Account Deployments in Terraform

Question:
Your Terraform script needs to create resources across multiple AWS accounts. How would you manage this securely?

Answer:
To manage cross-account deployments:

Use Multiple AWS Provider Configurations

```
provider "aws" {
  alias   = "account_a"
  region = "us-east-1"
  assume_role {
    role_arn = "arn:aws:iam::111111111111:role/TerraformRole"
  }
}
```

```
provider "aws" {
  alias  = "account_b"
  region = "us-west-2"
  assume_role {
    role_arn = "arn:aws:iam::222222222222:role/TerraformRole"
  }
}
```

1. **Assume IAM Roles Instead of Using Static Keys:**
   - Use IAM roles with trust relationships to allow Terraform execution from a central account.
2. **Use Separate State Files for Each Account:**
   - Store state files in different S3 buckets to avoid conflicts.
3. **Use Workspaces or Separate Environments:**
   - Create different Terraform workspaces for each account.

9️⃣Scenario: Terraform State File Corruption

Question:
Your Terraform state file got corrupted or lost. What steps would you take to recover it?

Answer:
To recover a lost or corrupted Terraform state file:

1. **Check Remote State Backup (if enabled):**

If using an S3 backend with versioning, restore the last known good state:

```
aws s3 cp s3://terraform-state-bucket/path/to/statefile.tfstate .
```

2. **Manually Reconstruct State with `terraform import`:**
   - If no backup is available, import existing resources into a new state file.

```
terraform import aws_instance.example i-1234567890abcdef0
```

3. **Use `terraform refresh`:**

Try to regenerate the state file from existing infrastructure:

```
terraform refresh
```

4. **Enable Backend State Locking for Future Safety:**

Use DynamoDB locking to prevent corruption:

```
dynamodb_table = "terraform-lock"
```

🔟 Scenario: Handling a Failed Terraform Deployment in CI/CD

Question:
You use Terraform in a CI/CD pipeline, but the deployment fails due to network issues. How do you handle failures and ensure deployments are reliable?

Answer:
To handle Terraform failures in CI/CD:

1. **Use `terraform plan` Before `apply`:**

Always validate changes before applying:

```
terraform plan -out=tfplan
```

2. **Enable Retries in Terraform:**

Use Terraform's retry mechanism:

```
retries = 3
retry_condition = "network_issue"
```

3. **Use a Remote State Backend:**
   - Ensure state consistency across pipeline runs by using AWS S3, Terraform Cloud, or GCP Storage.
4. **Auto-Recover Failed Resources:**

Implement `terraform taint` to force recreation of failed resources:

```
terraform taint aws_instance.example
terraform apply
```

5. **Implement Rollback Strategy:**
   - Use `terraform destroy` if deployment fails and needs a fresh start.

1️⃣1️⃣Scenario: Scaling Infrastructure Dynamically

Question:
Your application is experiencing high traffic, and you need to scale up EC2 instances dynamically using Terraform. How would you achieve this?

Answer:
To scale infrastructure dynamically with Terraform:

Use AWS Auto Scaling Groups (ASG):

```
resource "aws_autoscaling_group" "example" {
  desired_capacity     = 2
  max_size             = 5
  min_size             = 1
  launch_configuration = aws_launch_configuration.example.id
}
```

Use Terraform Variables for Scaling:

```
variable "max_instances" {
  default = 5
}
```

1. **Enable Auto Scaling Policies:**
   ○ Use AWS CloudWatch alarms to trigger scaling actions.
2. **Use Terraform Modules:**
   ○ Modularize the auto-scaling configuration for reusability.

1️⃣2️⃣Scenario: Preventing Costly Infrastructure Changes

Question:
A junior engineer accidentally ran `terraform apply` with incorrect values, leading to a costly cloud bill. How do you prevent such mistakes?

Answer:
To prevent unintended changes:

1. **Use Terraform Plan & Manual Approval:**

Implement a two-step workflow:

```
terraform plan -out=tfplan
terraform apply tfplan
```

○

Use `lifecycle` Block to Prevent Destruction:

```

resource "aws_instance" "example" {
  lifecycle {
    prevent_destroy = true
  }
}
```

2. **Restrict Terraform Execution:**

  - Use IAM policies to restrict who can run `terraform apply`.
3. **Use Sentinel or Policy as Code:**
   - Enforce policies using Terraform Cloud's Sentinel.
4. **Enable Cost Alerts:**
   - Set up AWS/GCP budget alerts to detect unexpected cost spikes.

1️⃣3️⃣Scenario: Resolving Drift Between Terraform State and Cloud Resources

Question:
You notice that some AWS EC2 instances were manually modified outside of Terraform, causing a drift. How would you detect and resolve this?

Answer:
To detect and resolve drift:

Use `terraform plan` to Check for Drift:

```
terraform plan
```

This compares the actual infrastructure with the state file and shows discrepancies.

Use `terraform state list` to Identify Changes:

```
terraform state list
```

Use `terraform refresh` to Sync State (if non-destructive):

```
terraform refresh
```

Use `terraform import` for Untracked Resources:

```
terraform import aws_instance.example i-1234567890abcdef0
```

Use `terraform taint` to Recreate Resources if Necessary:

```
terraform taint aws_instance.example
terraform apply
```

Enforce Infrastructure as Code:

- ○ Implement IAM policies to restrict manual changes.
- ○ Use Terraform Cloud with Sentinel for policy enforcement.

1️⃣4️⃣Scenario: Terraform Deployment Fails Due to Resource Name Conflicts

Question:
Your Terraform deployment fails with an error saying that an S3 bucket name already exists. How do you resolve this?

Answer:
S3 bucket names must be globally unique. To resolve:

Use a Unique Naming Convention with Variables:

```
resource "aws_s3_bucket" "example" {
  bucket = "my-bucket-${random_id.suffix.hex}"
}
```

```
resource "random_id" "suffix" {
  byte_length = 4
}
```

Check for Existing Buckets in AWS CLI:

```
aws s3 ls | grep my-bucket
```

Manually Delete Conflicting Bucket (If Allowed):

```
aws s3 rb s3://my-bucket --force
```

Use a Randomized Prefix Instead of Static Names:

Generate a unique prefix for resource names using:

```
bucket = "dev-${var.environment}-myapp"
```

1️⃣5️⃣Scenario: Managing Multiple Environments (Dev, Staging, Prod) with Terraform

Question:
Your company needs to maintain separate Terraform configurations for **Dev, Staging, and Prod** environments. How would you structure this?

Answer:

Use Terraform Workspaces:

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

Reference workspace in the configuration:

```
variable "env" {
  default = terraform.workspace
}
```

Use Separate State Files for Each Environment:

Store state in different S3 buckets:

```
backend "s3" {
  bucket = "terraform-state-${var.environment}"
  key    = "state/terraform.tfstate"
  region = "us-east-1"
}
```

Use Terraform Modules for Reusability:

- ○ Create a `modules` directory and use it in multiple environments.

```
module "networking" {
  source  = "../modules/networking"
  vpc_id  = var.vpc_id
}
```

2. **Use Git Branching for Environment Isolation:**
   - ○ Maintain `dev`, `staging`, and `prod` branches in Git.

1️⃣6️⃣Scenario: Handling Sensitive Data in Terraform

Question:
Your Terraform configuration requires storing database passwords and API keys. How do you securely manage sensitive data?

Answer:

Use Environment Variables Instead of Hardcoding:

```
export TF_VAR_db_password="mysecretpassword"
```

Reference in Terraform:

```
variable "db_password" {}
```

Use Terraform Vault Provider for Secrets Management:

Store secrets in HashiCorp Vault and retrieve them dynamically.

```
provider "vault" {
  address = "https://vault.example.com"
}
```

Use AWS Secrets Manager for Secure Storage:

Retrieve secrets securely in Terraform:

```
data "aws_secretsmanager_secret" "db_password" {
  name = "db_password"
}
```

Encrypt State File if Using Local Backend:

```
encrypt = true
```

Use `.gitignore` to Prevent Storing Sensitive Data in Git:

```
*.tfstate
*.tfvars
```

1️⃣7️⃣Scenario: Handling a Partial Failure in Terraform Apply

Question:
Your `terraform apply` ran successfully for some resources but failed for others. How do you fix this?

Answer:

Check Which Resources Were Created:

```
terraform state list
```

Retry Only the Failed Resources:

```
terraform apply -auto-approve
```

Use `terraform taint` if a Resource Is Partially Created:

```
terraform taint aws_instance.example
terraform apply
```

Destroy and Recreate Only the Failed Resources:

```
terraform destroy -target=aws_instance.example
terraform apply
```

Enable Checkpoints for Large Deployments:

Use `terraform apply -parallelism=1` for better debugging.

1️⃣8️⃣Scenario: Migrating Terraform State to a New Backend

Question:
Your team needs to migrate the Terraform state from local storage to an **AWS S3 backend**. How would you perform this migration safely?

Answer:

Configure the Remote S3 Backend in `backend.tf`:

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

Run Terraform Init with Migration Option:

```
terraform init -migrate-state
```

Verify That the State Is Stored in S3:

```
aws s3 ls s3://my-terraform-state/
```

Enable DynamoDB for State Locking:

```
resource "aws_dynamodb_table" "terraform_locks" {
  name           = "terraform-lock"
  billing_mode   = "PAY_PER_REQUEST"
  hash_key       = "LockID"
  attribute {
    name = "LockID"
    type = "S"
  }
}
```

1️⃣9️⃣ Scenario: Terraform Deployment with Zero Downtime

Question:
Your team wants to deploy an updated version of an application **without downtime** using Terraform. How would you achieve this?

Answer:

Use Blue-Green Deployment with an ALB:

Deploy the new version in a separate target group.

```
resource "aws_lb_target_group" "blue" { ... }
resource "aws_lb_target_group" "green" { ... }
```

Switch the ALB listener to the new target group.

Use Rolling Updates in Auto Scaling Groups:

```
min_elb_capacity = 2
max_elb_capacity = 5
```

Use `terraform apply` with `-parallelism=1` for Safe Updates:

```
terraform apply -parallelism=1
```

Implement Feature Flags to Control Traffic Routing.

→ TRAININGS

WE ARE DIFFERENT

At CareerByteCode, we redefine training by focusing on real-world, hands-on experience. Unlike traditional learning methods, we provide step-by-step implementation guides, 500+ real-time use cases, and industry-relevant projects across cutting-edge technologies like AWS, Azure, GCP, DevOps, AI, FullStack Development and more.

Our approach goes beyond theoretical knowledge—we offer expert mentorship, helping learners understand how to study effectively, close career gaps, and

16+
Years of operations

91+
Countries worldwide

241 K Happy clients

🌐 **Our Usecases Platform**
https://careerbytecode.substack.com

🌐 **Our WebShop**

CareerByteCode
Learning Made simple

CareerByteCode
All in One Platform

STAY IN TOUCH WITH US!

Website

Our WebShop https://careerbytecode.shop
Our Usecases Platform https://careerbytecode.substack.com

E-mail

careerbytec@gmail.com

HQ address

Belgium, Europe

Social Media

@careerbytecode

Phone

+32 471 40 8908

For any RealTime Handson Projects

And for more tips like this

+ Follow

Like & ReShare

in @techopsbysonali