Here are the **55 scenario-based Terraform Associate practice questions** covering the exam areas, including correct answers and explanations.

---

1. Understanding Terraform Basics

Question 1:
You are tasked with managing infrastructure for a company that uses multiple environments (dev, staging, and prod). Each environment has slightly different configurations. How would you efficiently manage the infrastructure using Terraform to ensure consistency and scalability?

A. Create a separate Terraform configuration file for each environment.
B. Use a single Terraform configuration file with `if` statements to define resources for each environment.
C. Use workspaces to manage multiple environments within a single configuration.
D. Create separate Terraform projects for each environment and manually copy changes between them.

Answer: C
Explanation: Terraform workspaces allow you to manage multiple environments within a single configuration. This approach ensures consistency and scalability, as all environments share the same configuration but differ by state.

---

2. Managing Providers and Resources

Question 2:
You want to deploy an application on AWS using Terraform, and you need to specify a provider for AWS. Which of the following steps should you take to define the provider in your Terraform configuration?

A. Use the `provider` block to specify the AWS region.
B. Use the `resource` block directly to define resources without specifying a provider.
C. Use a backend configuration to define the provider.
D. Use the `module` block to define the AWS provider.

Answer: A
Explanation: The `provider` block is used to configure the specific provider details, such as region for AWS. Terraform uses this to authenticate and communicate with the provider's API.

---

3. Terraform State

Question 3:
Your team is collaboratively working on a Terraform project. You notice that team members are overwriting each other's changes in the state file. What is the best solution to prevent this issue?

A. Use `terraform plan` before applying changes to identify potential conflicts.
B. Store the state file in a version control system like Git.
C. Use a remote backend with state locking, such as AWS S3 with DynamoDB.
D. Disable state management to avoid conflicts.

Answer: C
Explanation: A remote backend with state locking ensures that only one user can modify the state at a time. AWS S3 with DynamoDB for state locking is a commonly used solution.

---

4. Data Sources in Terraform

Question 4:
Your organization has an existing VPC in AWS, and you need to use it in your Terraform configuration without recreating it. Which feature of Terraform allows you to achieve this?

A. Use a `resource` block to define the VPC in the configuration.
B. Use a `provider` block to fetch the VPC details.
C. Use a `data` block to retrieve information about the existing VPC.
D. Use `terraform import` to directly import the VPC.

Answer: C
Explanation: The `data` block allows you to retrieve and use information about existing resources managed outside of Terraform.

---

5. Modules in Terraform

Question 5:
You are working on a Terraform project that involves deploying a complex architecture with multiple reusable components. How can you organize your Terraform code for reusability?

A. Write a single large Terraform configuration file.
B. Create multiple smaller Terraform configuration files and hardcode values.
C. Use Terraform modules to encapsulate reusable logic.
D. Duplicate the configuration files for each component.

Answer: C
Explanation: Modules encapsulate reusable logic, making it easier to organize and maintain complex configurations. They promote reusability and reduce code duplication.

---

6. Terraform CLI Commands

Question 6:
A junior team member ran `terraform apply` without using `terraform plan` and caused unintended changes. How can you enforce reviewing the execution plan before applying changes in the future?

A. Use version control to prevent unplanned changes.
B. Use the `-auto-approve` flag with `terraform apply`.
C. Require the team to use `terraform plan` before `terraform apply`.
D. Integrate `terraform apply` with a CI/CD pipeline that enforces plan review.

Answer: D
Explanation: Enforcing plan review through a CI/CD pipeline ensures that changes are reviewed and approved before they are applied, reducing the risk of unintended changes.

---

7. Remote State

Question 7:
Your Terraform state file is stored remotely in an S3 bucket. To ensure consistency, you want to lock the state file when applying changes. What additional service should you configure?

A. AWS CloudTrail
B. AWS DynamoDB
C. AWS IAM
D. AWS Lambda

Answer: B
Explanation: AWS DynamoDB is used for state locking when the state file is stored in an S3 bucket. This prevents multiple users from simultaneously modifying the state.

---

8. Terraform Syntax and HCL

Question 8:
You want to conditionally create an AWS EC2 instance only if a variable `create_instance` is set to `true`. Which syntax would you use in the Terraform configuration?

A. Use `count = var.create_instance ? 1 : 0` in the resource block.
B. Use an `if` statement in the Terraform configuration.
C. Use `for_each` to iterate over the variable.
D. Use a `data` block to fetch the instance details.

Answer: A
Explanation: The `count` argument is used for conditional resource creation. If `create_instance` is `true`, `count` is set to 1, otherwise it is set to 0, effectively skipping resource creation.

---

9. Terraform Import

Question 9:
Your team is managing infrastructure manually but wants to start using Terraform. You decide to import an existing AWS EC2 instance into Terraform. What command would you use?

A. `terraform init`
B. `terraform import aws_instance.my_instance i-1234567890abcdef0`
C. `terraform plan`
D. `terraform apply`

Answer: B
Explanation: The `terraform import` command is used to bring existing resources into Terraform's state, associating them with a resource block in your configuration.

---

10. Managing Drift

Question 10:
You suspect that manual changes were made to your infrastructure outside of Terraform. What is the best way to detect and resolve this drift?

A. Use `terraform apply` to automatically fix any drift.
B. Use `terraform refresh` to sync the state file with the real infrastructure.
C. Use `terraform validate` to check for drift.
D. Manually inspect the infrastructure for changes.

Answer: B
Explanation: The `terraform refresh` command updates the state file to reflect the current state of the infrastructure, helping detect any drift.

---

11. Terraform Backend Configuration

Question 11:
Your team uses a shared S3 bucket to store Terraform state files. What additional

configuration is required to ensure multiple users do not overwrite the state file when making changes simultaneously?

A. Configure an S3 bucket policy for write permissions.
B. Enable versioning on the S3 bucket.
C. Configure a DynamoDB table for state locking.
D. Use local backends instead of remote backends.

Answer: C
Explanation: A DynamoDB table is used for state locking when using an S3 backend. This ensures that only one user can make changes to the state file at a time, preventing concurrency issues.

---

12. Terraform Plan and Apply

Question 12:
Your Terraform configuration includes several resources. You need to apply only the changes related to a specific resource without affecting others. How can you achieve this?

A. Use `terraform apply` without any additional flags.
B. Use the `-auto-approve` flag with `terraform apply`.
C. Use the `-target` flag with `terraform apply` to specify the resource.
D. Use `terraform plan` and manually delete unrelated resources from the plan output.

Answer: C
Explanation: The `-target` flag in Terraform allows you to specify a particular resource or module to apply changes to without affecting other resources.

---

13. Terraform Providers

Question 13:
You are managing infrastructure on both AWS and Azure. How do you configure Terraform to interact with both providers within the same configuration?

A. Use separate Terraform projects for AWS and Azure.
B. Define multiple `provider` blocks in a single configuration.
C. Use workspaces to switch between AWS and Azure.
D. Use a single `provider` block with multiple regions.

Answer: B
Explanation: Multiple `provider` blocks allow you to configure and use multiple cloud providers (like AWS and Azure) within the same Terraform configuration.

---

14. Input Variables

Question 14:
You are creating a Terraform configuration that should accept user inputs for values like `instance_type` and `region`. What is the best way to define and use these inputs?

A. Use environment variables to pass inputs to the Terraform configuration.
B. Use hardcoded values in the configuration file.
C. Define variables in a `variables.tf` file and use `terraform.tfvars` to pass values.
D. Prompt users to enter values manually during `terraform apply`.

Answer: C
Explanation: The best practice is to define input variables in a `variables.tf` file and provide their values in a `terraform.tfvars` file. This approach keeps the configuration flexible and maintainable.

---

15. Output Values

Question 15:
You need to provide the public IP address of an AWS EC2 instance created by Terraform to another team. How can you make this information easily accessible?

A. Share the Terraform state file with the team.
B. Use the `output` block in your configuration to export the public IP.
C. Write the public IP to an external file using a script.
D. Manually fetch the IP from the AWS Console.

Answer: B
Explanation: The `output` block in Terraform is used to expose specific attributes of resources to the end user. This makes it easy to share critical information like the public IP.

---

16. Terraform Provisioners

Question 16:
You need to execute a script on an AWS EC2 instance after it is created. Which feature of Terraform should you use?

A. Use the `data` block to fetch script details.
B. Use `provisioner "local-exec"` to execute the script on the instance.
C. Use `provisioner "remote-exec"` to run the script on the instance.
D. Use the `output` block to expose the script details.

Answer: C
Explanation: The `provisioner "remote-exec"` is used to execute scripts or commands on a remote resource, such as an EC2 instance, after it is provisioned.

---

17. Dependency Management

Question 17:
In your configuration, an AWS Lambda function needs to access an S3 bucket. How do you ensure Terraform creates the S3 bucket before the Lambda function?

A. Use the `depends_on` argument in the Lambda resource block.
B. Define the Lambda function and S3 bucket in separate files.
C. Manually run `terraform apply` for the S3 bucket before the Lambda function.
D. Use `terraform refresh` to update the state.

Answer: A
Explanation: The `depends_on` argument explicitly defines resource dependencies, ensuring the S3 bucket is created before the Lambda function.

---

18. Workspaces

Question 18:
You manage multiple environments (dev, staging, prod) using Terraform. What is the best way to isolate state files for each environment while using the same configuration?

A. Create separate Terraform projects for each environment.
B. Use Terraform workspaces to isolate state files.
C. Manually copy state files for each environment.
D. Use different backend configurations for each environment.

Answer: B
Explanation: Terraform workspaces provide a mechanism for isolating state files within a single configuration. This is ideal for managing multiple environments.

---

19. Handling Sensitive Data

Question 19:
You need to manage sensitive data like passwords and API keys in your Terraform configuration. What is the recommended approach?

A. Store sensitive data directly in the configuration file.
B. Use Terraform variables with the `sensitive = true` attribute.

C. Hardcode sensitive values in `terraform.tfvars`.
D. Use an external secrets manager like AWS Secrets Manager or HashiCorp Vault.

Answer: D
Explanation: Storing sensitive data in a secrets manager ensures better security and avoids exposing sensitive values in your Terraform configuration or state files.

---

20. Terraform State Manipulation

Question 20:
A resource was manually deleted in your cloud provider's console, but it still exists in the Terraform state file. How can you update the state to reflect this change?

A. Run `terraform refresh` to update the state file.
B. Use `terraform plan` to sync the state.
C. Use `terraform import` to re-import the resource.
D. Use `terraform state rm` to remove the resource from the state file.

Answer: D
Explanation: The `terraform state rm` command removes a resource from the state file. This is useful when the resource no longer exists in the real infrastructure.

---

21. Resource Configuration with `count`

Question 21:
You need to create multiple AWS EC2 instances for a test environment, where the number of instances is determined by the value of a variable `instance_count`. How should you configure the resource block?

A. Use a `for_each` loop with a map of instances.
B. Use `count = var.instance_count` in the resource block.
C. Create separate resource blocks for each instance.
D. Use `depends_on` to dynamically create instances.

Answer: B
Explanation: The `count` argument allows you to create multiple instances of a resource dynamically based on the value of a variable. This is ideal for scenarios where the number of resources is determined at runtime.

---

22. Terraform Outputs

Question 22:
You are working on a Terraform configuration that creates multiple AWS resources. The output includes sensitive values like access keys, which should not be displayed in the CLI. How can you hide these sensitive values?

A. Omit the sensitive outputs from the `output` block.
B. Use the `sensitive = true` attribute in the `output` block.
C. Store the outputs in a file instead of displaying them.
D. Use an environment variable to mask sensitive outputs.

Answer: B
Explanation: The `sensitive = true` attribute in the `output` block prevents Terraform from displaying sensitive values in the CLI output, protecting sensitive information.

---

23. Modules with Variables

Question 23:
You are reusing a Terraform module that requires specific inputs. How do you pass variables to the module?

A. Use a `data` block to fetch variables.
B. Define variables in the root module and reference them in the child module.
C. Use `terraform.tfvars` to automatically pass variables to the module.
D. Use the `module` block and pass variables as arguments.

Answer: D
Explanation: Variables are passed to a module using the `module` block, where you define arguments matching the variable names in the module.

---

24. Dependency Graph

Question 24:
You need to ensure that Terraform creates an IAM role before attaching the policy. How does Terraform determine the correct order of operations?

A. By defining resources in the correct order in the configuration file.
B. Using the `depends_on` argument explicitly.
C. By analyzing implicit dependencies based on resource references.
D. By using a pre-defined execution plan.

Answer: C
Explanation: Terraform automatically determines the correct order of operations by analyzing resource references and implicit dependencies in the configuration.

---

25. Terraform Debugging

Question 25:
Your Terraform configuration is throwing an error, and you need detailed logs to debug the issue. What should you do?

A. Use the `terraform refresh` command to identify the error.
B. Set the `TF_LOG` environment variable to `DEBUG` and re-run the command.
C. Run `terraform validate` to fix the error.
D. Manually inspect the configuration file for errors.

Answer: B
Explanation: Setting the `TF_LOG` environment variable to `DEBUG` provides detailed logs, which are useful for troubleshooting issues in Terraform.

---

26. Providers with Multiple Configurations

Question 26:
You need to deploy resources in two different AWS regions using Terraform. How can you achieve this?

A. Use a single `provider` block with region interpolation.
B. Define two separate `provider` blocks with aliases for the regions.
C. Use Terraform workspaces to switch between regions.
D. Manually deploy resources for each region.

Answer: B
Explanation: You can define multiple `provider` blocks with aliases for each region. Use the `provider` argument in resources to specify which provider configuration to use.

---

27. Terraform Data Sources

Question 27:
Your configuration requires the latest AMI ID for a specific instance type in AWS. How can you dynamically retrieve this value?

A. Use the `resource` block to create the AMI.
B. Use the `provider` block to specify the AMI ID.
C. Use a `data` block to query the latest AMI ID.
D. Use `terraform import` to import the AMI ID.

Answer: C
Explanation: The `data` block allows you to query existing infrastructure information, such as the latest AMI ID, dynamically at runtime.

---

28. Terraform Import

Question 28:
You want to import an existing AWS S3 bucket into your Terraform state. What steps are required to complete this process?

A. Add the S3 bucket configuration in your Terraform file, then run `terraform import`.
B. Run `terraform init`, and the bucket will be imported automatically.
C. Use `terraform plan` to import the bucket.
D. Manually edit the state file to include the bucket.

Answer: A
Explanation: To import an existing resource, define it in the configuration file and then use the `terraform import` command to associate it with the Terraform state.

---

29. Terraform Validation

Question 29:
Before deploying your configuration, you want to ensure it is syntactically correct. What command should you use?

A. `terraform fmt`
B. `terraform validate`
C. `terraform plan`
D. `terraform apply`

Answer: B
Explanation: The `terraform validate` command checks your configuration for syntax errors and validates the structure before applying it.

---

30. Remote Backends

Question 30:
You want to store Terraform state files securely and share them across your team. Which backend should you configure?

A. Local backend
B. GitHub backend

C. S3 backend with DynamoDB for locking
D. Terraform Enterprise

Answer: C
Explanation: The S3 backend with DynamoDB locking is a secure and scalable solution for storing and sharing Terraform state files, preventing simultaneous modifications.

---

31. Dynamic Blocks

Question 31:
You are configuring multiple security group rules for an AWS Security Group. Instead of writing multiple resource blocks, how can you simplify the configuration?

A. Use a `count` argument in the resource block.
B. Use a `dynamic` block within the resource.
C. Use multiple `provider` blocks for each rule.
D. Write separate resource blocks for each rule.

Answer: B
Explanation: A `dynamic` block generates multiple nested blocks dynamically, based on input variables or other conditions.

---

32. Workspace Isolation

Question 32:
Your team is working on multiple projects in Terraform. Each team member accidentally uses the wrong state file. How can you enforce workspace-specific state isolation?

A. Use separate state files manually managed by the team.
B. Configure workspaces for each project using `terraform workspace`.
C. Use environment variables to switch states.
D. Use Terraform modules to manage state files.

Answer: B
Explanation: Workspaces are a built-in feature in Terraform to isolate state files for different projects or environments.

---

33. State Locking

Question 33:
Your team has encountered a situation where two engineers are attempting to apply

changes to the same Terraform state file simultaneously. How can you prevent this from happening?

A. Use Terraform workspaces to isolate changes.
B. Enable state locking by configuring a backend with locking support, such as S3 with DynamoDB.
C. Use `terraform plan` to check for potential conflicts.
D. Manually coordinate changes among the team members.

Answer: B
Explanation: Backends like S3 with DynamoDB or Terraform Cloud support state locking, ensuring that only one operation can modify the state at a time, preventing conflicts.

---

34. Terraform Cloud Variables

Question 34:
Your organization uses Terraform Cloud to manage infrastructure. How can you store sensitive variables like API keys securely in Terraform Cloud?

A. Store the sensitive variables in the `variables.tf` file.
B. Use environment variables on local machines.
C. Mark the variable as sensitive when defining it in Terraform Cloud.
D. Use a separate state file to manage sensitive variables.

Answer: C
Explanation: Terraform Cloud provides an option to mark variables as sensitive. This prevents the sensitive value from being displayed in logs or outputs.

---

35. Custom Provider Configuration

Question 35:
You are working with a custom Terraform provider for an internal application. How can you install and use the custom provider in your configuration?

A. Use the `terraform init` command, and Terraform will automatically install custom providers.
B. Place the provider binary in the Terraform `plugins` directory and run `terraform init`.
C. Modify the `terraform-provider` binary directly.
D. Use the `terraform plan` command to initialize the provider.

Answer: B
Explanation: For custom providers, you need to place the binary in the correct directory (e.g., `.terraform/plugins`) and then run `terraform init` to initialize it.

---

36. Resource Lifecycle Management

Question 36:
You need to ensure that a particular resource is destroyed before creating a new one in your Terraform configuration. Which argument can you use to manage this dependency?

A. `depends_on`
B. `count`
C. `lifecycle { create_before_destroy = true }`
D. `provisioner`

Answer: C
Explanation: The `lifecycle` block with the `create_before_destroy = true` argument ensures Terraform creates a new resource before destroying the old one, reducing downtime.

---

37. File Interpolation

Question 37:
You want to dynamically load content from a file and use it as a resource argument in your Terraform configuration. Which function should you use?

A. `file()`
B. `templatefile()`
C. `join()`
D. `replace()`

Answer: A
Explanation: The `file()` function reads the contents of a file and returns it as a string, which can be used in your Terraform configuration.

---

38. Handling State File Corruption

Question 38:
Your Terraform state file was accidentally corrupted. How can you recover the state and resume managing your infrastructure?

A. Manually edit the state file to fix errors.
B. Use the `terraform state push` command with a backup file.
C. Recreate the resources to generate a new state file.
D. Use `terraform plan` to synchronize the state.

Answer: B
Explanation: The `terraform state push` command can restore the state using a backup file, helping you recover from state file corruption.

---

39. Policy as Code

Question 39:
Your organization requires that all Terraform configurations comply with specific policies, such as tagging all resources. How can you enforce these policies?

A. Use environment variables to enforce policies.
B. Define the policies in a separate configuration file and include it in your code.
C. Use Sentinel or third-party tools to enforce policies during plan and apply stages.
D. Perform manual reviews of the Terraform configuration.

Answer: C
Explanation: Sentinel, a policy-as-code framework by HashiCorp, allows you to enforce policies during `terraform plan` and `terraform apply` stages, ensuring compliance.

---

40. Destroy Commands

Question 40:
You need to destroy a specific resource without affecting the rest of the infrastructure. How can you achieve this?

A. Use `terraform destroy` without specifying the resource.
B. Use the `-target` flag with `terraform destroy` to specify the resource.
C. Manually delete the resource in the cloud provider console.
D. Use `terraform plan` and remove the resource from the state file.

Answer: B
Explanation: The `-target` flag with `terraform destroy` allows you to specify and destroy only the desired resource while leaving others intact.

---

41. Managing State Drift

Question 41:
Your infrastructure has drifted from the Terraform state file due to manual changes. How can you update the state file to reflect the actual infrastructure?

A. Run `terraform plan` to synchronize the state file.
B. Use `terraform refresh` to update the state file with current resource attributes.

C. Use `terraform state rm` to remove the drifted resources.
D. Recreate the state file from scratch.

Answer: B
Explanation: The `terraform refresh` command updates the Terraform state file to reflect the actual current state of the infrastructure.

---

42. Dynamic Provider Credentials

Question 42:
You want Terraform to use AWS credentials dynamically based on the environment. How can you configure this?

A. Use hardcoded credentials in the provider block.
B. Use environment variables like `AWS_ACCESS_KEY_ID` and `AWS_SECRET_ACCESS_KEY`.
C. Use the `default` profile in the AWS CLI configuration.
D. Use `terraform apply` to pass credentials interactively.

Answer: B
Explanation: Terraform can dynamically use AWS credentials from environment variables like `AWS_ACCESS_KEY_ID` and `AWS_SECRET_ACCESS_KEY`, providing flexibility across environments.

---

43. Remote Execution

Question 43:
How does Terraform Cloud perform remote execution of plans and applies?

A. It executes plans and applies locally on the user’s machine.
B. It uses agents to execute the operations on remote servers.
C. It executes plans and applies on Terraform Cloud's infrastructure.
D. It requires users to download and run the operations manually.

Answer: C
Explanation: Terraform Cloud executes plans and applies on its managed infrastructure, providing a centralized, collaborative workflow for teams.

---

44. Output Dependency

Question 44:
You need to pass the output of one Terraform configuration to another. How can you achieve this?

A. Use `terraform output` to export the value and manually pass it to the next configuration.
B. Use the `remote_state` data source to retrieve the output.
C. Use an external tool to share outputs between configurations.
D. Write the output to a file and read it in the second configuration.

Answer: B
Explanation: The `remote_state` data source allows one Terraform configuration to access the outputs of another, enabling seamless integration between configurations.

---

45. Terraform Workspaces Use Case

Question 45:
You are managing multiple environments (dev, staging, and production) using Terraform. How can you manage these environments without duplicating the configuration files?

A. Use multiple `.tf` files for each environment.
B. Use Terraform workspaces to switch between environments.
C. Use `count` to create resources for all environments in the same configuration.
D. Create separate state files for each environment manually.

Answer: B
Explanation: Terraform workspaces allow you to maintain a single configuration and isolate state files for different environments like dev, staging, and production.

---

46. Parallelism in Terraform

Question 46:
You have a large infrastructure deployment and want Terraform to provision resources faster. How can you optimize the deployment process?

A. Increase the parallelism level using the `-parallelism` flag in `terraform apply`.
B. Use multiple `terraform apply` commands simultaneously.
C. Disable locking to allow faster operations.
D. Manually create resources in parallel in the provider's console.

Answer: A
Explanation: The `-parallelism` flag controls the number of resources Terraform creates simultaneously. Increasing this value can speed up deployments but should be used cautiously to avoid overwhelming the provider's API.

---

47. Sensitive Data in Terraform

Question 47:
How can you prevent sensitive data (e.g., passwords or secrets) from being displayed in plain text in the Terraform state file?

A. Use environment variables to pass sensitive data.
B. Mark variables as `sensitive = true` in the variable block.
C. Encrypt the state file manually after each update.
D. Store sensitive data in an external tool like HashiCorp Vault and retrieve it dynamically.

Answer: D
Explanation: Storing sensitive data in an external secret management tool like HashiCorp Vault ensures that Terraform does not store sensitive values in plain text in the state file.

---

48. Remote Backends with Version Control

Question 48:
You want to ensure Terraform state is stored centrally and locked during operations while keeping the configuration files versioned in Git. Which setup should you use?

A. Use the local backend and commit the state file to the repository.
B. Use an S3 backend with DynamoDB for locking and Git for versioning configuration files.
C. Use Terraform Cloud for both state storage and configuration management.
D. Use a remote backend to store state and a remote Git repository for configuration.

Answer: B
Explanation: Using an S3 backend with DynamoDB provides centralized and locked state storage, while Git ensures the configuration files are version-controlled separately.

---

49. Conditional Expressions

Question 49:
You want to conditionally create an AWS resource based on the value of a variable `create_resource`. How would you achieve this?

A. Use a `count` argument with a conditional expression like `count = var.create_resource ? 1 : 0`.
B. Use a `for_each` loop to iterate conditionally.
C. Add a `depends_on` block with the variable.
D. Create the resource manually if the variable is true.

Answer: A
Explanation: The `count` argument supports conditional expressions, allowing you to conditionally create resources based on the value of a variable.

---

50. Terraform depends_on Argument

Question 50:
In which scenario should you use the `depends_on` argument explicitly in a Terraform resource block?

A. To create a resource before others based on implicit dependencies.
B. When the dependency is not automatically inferred by Terraform.
C. To define a dependency between resources that have no shared attributes.
D. Both B and C.

Answer: D
Explanation: The `depends_on` argument is used when Terraform cannot infer dependencies automatically, such as when there is no direct attribute relationship between resources.

---

51. Terraform Version Constraints

Question 51:
How can you ensure that a Terraform configuration only works with specific Terraform versions (e.g., 1.1.0 or higher but not 2.x)?

A. Specify the version in the `provider` block.
B. Add a `required_version` constraint in the `terraform` block.
C. Use a specific Terraform binary for execution.
D. Define the version in a `.terraformrc` file.

Answer: B
Explanation: The `required_version` argument in the `terraform` block specifies version constraints, ensuring compatibility with certain Terraform versions.

---

52. Preventing Resource Drift

Question 52:
How can you ensure that a resource managed by Terraform does not drift from the desired configuration?

A. Run `terraform refresh` regularly.
B. Use the `lifecycle { prevent_destroy = true }` argument.
C. Use the `terraform apply` command frequently.
D. Regularly audit and update the configuration files.

Answer: A
Explanation: The `terraform refresh` command updates the Terraform state to match the actual infrastructure, identifying and preventing resource drift.

---

53. Handling State Migration

Question 53:
You want to move the Terraform state from the local backend to an S3 backend. What is the correct sequence of steps?

A. Modify the backend block and run `terraform init` with `-reconfigure`.
B. Directly edit the state file and upload it to S3.
C. Copy the state file manually to the S3 bucket.
D. Run `terraform state push` to migrate the state file.

Answer: A
Explanation: Updating the backend block and running `terraform init` with `-reconfigure` migrates the state file from the local backend to the new S3 backend.

---

54. Terraform `null_resource` Use Case

Question 54:
When would you use the `null_resource` resource in a Terraform configuration?

A. To manage resources that do not have a Terraform provider.
B. To execute local commands or provisioners without creating any actual resources.
C. To define placeholder resources for future use.
D. To test configurations without applying them.

Answer: B
Explanation: The `null_resource` is used to trigger provisioners or local commands without creating actual resources, making it useful for tasks like configuration management.

---

55. Terraform CLI Automation

Question 55:
You want to automate Terraform operations in a CI/CD pipeline. Which approach is best for securely passing variables?

A. Hardcode the variables in the configuration.
B. Use a `.tfvars` file stored in the repository.

C. Pass the variables as environment variables in the CI/CD pipeline.
D. Include variables directly in the pipeline script.

Answer: C
Explanation: Using environment variables in the CI/CD pipeline ensures variables are passed securely and are not exposed in the code or configuration files.