CertyIQ


# CertyIQ

## Premium exam material

Get certification quickly with the CertyIQ Premium exam material.
Everything you need to prepare, learn & pass your certification exam easily. Lifetime free updates
First attempt guaranteed success.

https://www.CertyIQ.com

# About CertyIQ

We here at CertyIQ eventually got enough of the industry's greedy exam paid for. Our team of IT professionals comes with years of experience in the IT industry Prior to training CertIQ we worked in test areas where we observed the horrors of the paywall exam preparation system.

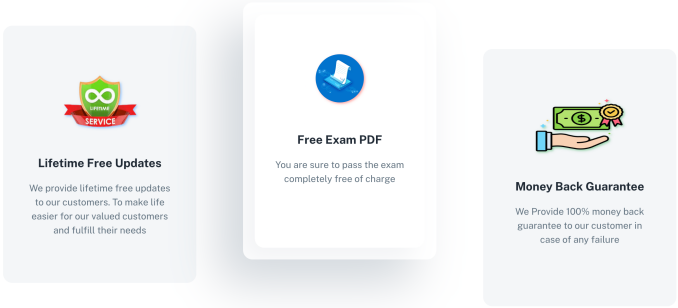
The misuse of the preparation system has left our team disillusioned. And for that reason, we decided it was time to make a difference. We had to make In this way, CertyIQ was created to provide quality materials without stealing from everyday people who are trying to make a living.

# Doubt Support

We have developed a very scalable solution using which we are able to solve 400+ doubts every single day with an average rating of 4.8 out of 5.

https://www.certyiq.com

Mail us on - certyiqofficial@gmail.com




**John**

October 19, 2022

★★★★★

Thanks you so much for your help. I scored 972 in my exam today. More than 90% were from your PDFs!

**Dana**

September 04, 2022

★★★★★

Thanks a lot for this updated AZ-900 Q&A. I just passed my exam and got 974, I followed both of your Az-900 videos and the 6 PDF, the PDFs are very much valid, all answers are correct. Could you please create a similar video/PDF for DP900, your content/PDF's is really awesome. The team did a really good job. Thank You 😊.

**Ahamed Shibly**

2 months ago

★★★★★

Customer support is realy fast and helpful, I just finished my exam and this video along with the 6 PDF helped me pass! Definitely recommend getting the PDFs. Thank you!

October 22, 2022

★★★★★

Passed my exam today with 891 marks. Out of 52 questions, 51 were from certyiq PDFs including Contoso case study.
Thank You certyiq team!

**Henry Rome**

2 months ago

★★★★★

These questions are real and 100 % valid. Thank you so much for your efforts, also your 4 PDFs are awesome, I passed the DP900 exam on 1 Sept. With 968 marks. Thanks a lot, buddy!

**Esmaria**

2 months ago

★★★★★

Simple easy to understand explanations. To anyone out there wanting to write AZ900, I highly recommend 6 PDF's.Thank you so much, appreciate all your hard work in having such great content. Passed my exam Today - 3 September with 942 score.

# Hashicorp

(Terraform Associate)

HashiCorp Certified

Total: **351 Questions**

Link: https://certyiq.com/papers/hashicorp/terraform-associate

**Question: 1** CertyIQ

The terraform.tfstate file always matches your currently built infrastructure.

A. True
B. False

**Answer: B**

**Explanation:**

The statement "The terraform.tfstate file always matches your currently built infrastructure" is false. Terraform's state file (terraform.tfstate) is a crucial component, serving as a record of the resources Terraform manages, along with their configurations and IDs. However, it doesn't automatically synchronize with the real-world infrastructure. The state file is updated based on successful Terraform operations like apply. If changes occur outside of Terraform (e.g., manual changes via the cloud provider's console or API, or other automated tools), the state file will become out of sync. This discrepancy between the state file and the actual infrastructure is known as "drift".

Therefore, simply having a terraform.tfstate file doesn't guarantee it accurately reflects the current state of the resources. Terraform relies on the state file for planning and making informed decisions about changes. If the state file is not up-to-date, applying changes can lead to unintended consequences, such as resources being overwritten or destroyed incorrectly. To maintain accurate state, it's essential to only modify infrastructure using Terraform operations and to periodically refresh the state using terraform refresh to account for any external changes. Furthermore, it is good practice to store state remotely in a secure and versioned way which allows for collaboration and avoids data loss when shared between multiple users. The Terraform state is a snapshot of the configuration last applied through Terraform and has to be explicitly managed to ensure accuracy.

Refer to the following resources for more details:

**Terraform State Documentation:** https://www.terraform.io/docs/language/state/index.html
**Managing Terraform State:** https://www.hashicorp.com/blog/managing-terraform-state
**Terraform Refresh Command:** https://www.terraform.io/docs/cli/commands/refresh.html

**Question: 2** CertyIQ

One remote backend configuration always maps to a single remote workspace.

A. True
B. False

**Answer: B**

**Explanation:**

The statement "One remote backend configuration always maps to a single remote workspace" is false. A single remote backend configuration, defined within a Terraform project, does not necessarily dictate a one-to-one relationship with a single remote workspace. While it's common to use one backend configuration for one workspace, Terraform allows for the dynamic configuration of workspaces through variables and command-line arguments. This flexibility allows a single backend configuration to potentially interact with different workspaces depending on the chosen variable values or environment. For instance, using the -workspace flag during terraform init, plan, or apply commands, you can target different workspaces while retaining the same remote backend configuration. The same backend configuration (such as using a specific

Terraform Cloud organization and its associated api token), can be reused with multiple workspace names. This configuration instructs Terraform to interact with the backend, while the specific workspace it interacts with is determined through variables or flags. The workspace itself defines the state files, which differ between workspaces. Therefore, while a backend config connects to a backend, it does not dictate which state file (workspace) is used. This capability is crucial for managing multiple environments (development, staging, production) or independent project components within the same organization, efficiently using a single backend setup.

Authoritative link for further research:Terraform Remote StateTerraform WorkspacesBackend Configuration

**Question: 3** CertyIQ

How is the Terraform remote backend different than other state backends such as S3, Consul, etc.?

A. It can execute Terraform runs on dedicated infrastructure on premises or in Terraform Cloud
B. It doesn't show the output of a terraform apply locally
C. It is only available to paying customers
D. All of the above

**Answer: A**

**Explanation:**

The correct answer is A. The primary distinction of the Terraform remote backend, specifically when referring to Terraform Cloud or Enterprise, lies in its ability to execute Terraform runs remotely on dedicated infrastructure. This is a fundamental departure from backends like S3 or Consul, which primarily serve as state storage locations. These storage backends require local execution of Terraform commands by the user. In contrast, the remote backend offloads the execution to a controlled environment, whether that's HashiCorp's managed service or self-hosted infrastructure. This offers benefits like centralized execution, improved team collaboration, and consistent environments. Option B is incorrect because Terraform Cloud (the most common implementation of a remote backend) can show the output of a Terraform apply through a web interface, a CLI, or an API. Option C is incorrect as the Terraform remote backend's basic functionality is also available through Terraform Cloud's free tier.

Therefore, the key differentiator of the remote backend is the remote execution capability, not just storage. This fundamentally shifts how Terraform is used, allowing for improved infrastructure management practices. While storage backends like S3 or Consul handle state persistence, they do not provide compute resources for running Terraform code. The remote backend's integration with Terraform Cloud or Enterprise allows features like version control integration, policy enforcement, and secrets management, further distinguishing it from simpler state storage backends. The remote execution also supports the concept of collaborative infrastructure-as-code, because the remote backend manages the environment within which changes are performed. The ability to manage the infrastructure used for the execution provides consistent results and supports greater team collaboration.

Links for further research:

Terraform Backends Documentation
Terraform Cloud Documentation
Terraform Enterprise Documentation

**Question: 4** CertyIQ

What is the workflow for deploying new infrastructure with Terraform?

A. terraform plan to import the current infrastructure to the state file, make code changes, and terraform apply to update the infrastructure.

B. Write a Terraform configuration, run terraform show to view proposed changes, and terraform apply to create new infrastructure.

C. terraform import to import the current infrastructure to the state file, make code changes, and terraform apply to update the infrastructure.

D. Write a Terraform configuration, run terraform init, run terraform plan to view planned infrastructure changes, and terraform apply to create new infrastructure.

**Answer: D**

**Explanation:**

The correct workflow for deploying new infrastructure with Terraform is outlined in option D. It begins with writing a Terraform configuration, which defines the desired state of your infrastructure using HashiCorp Configuration Language (HCL). Next, terraform init initializes the working directory, downloading necessary provider plugins and configuring the backend for state storage. After initialization, terraform plan generates an execution plan, showing the changes Terraform will make to achieve the desired state. This plan allows you to review the proposed infrastructure modifications before implementation. Finally, terraform apply executes the plan, creating or modifying the infrastructure components as defined in your configuration. Option A incorrectly suggests using terraform plan for importing existing infrastructure, while terraform import is the appropriate command for that task (as mentioned in C). Option B misses the crucial terraform init step and incorrectly suggests using terraform show instead of terraform plan. The Terraform workflow prioritizes planning before applying changes, making option D the most accurate and safest practice. This phased approach ensures predictability and prevents unintended consequences when managing cloud infrastructure.

Further reading:

**Terraform Workflow:** https://developer.hashicorp.com/terraform/tutorials/aws-get-started/terraform-workflow
**Terraform Init Command:** https://developer.hashicorp.com/terraform/cli/commands/init
**Terraform Plan Command:** https://developer.hashicorp.com/terraform/cli/commands/plan
**Terraform Apply Command:** https://developer.hashicorp.com/terraform/cli/commands/apply

**Question: 5** CertyIQ

A provider configuration block is required in every Terraform configuration.
Example:

```
provider "provider_name" {
   ...
}
```

A. True

B. False

**Answer: B**

**Explanation:**

False. A provider configuration block is not required in every Terraform configuration. Provider configuration

blocks are only required when you are using a particular provider to interact with a specific type of infrastructure resource. If your configuration does not interact with any resources provided by external providers, then you do not need to include a provider configuration block.

**Question: 6** CertyIQ

You run a local-exec provisioner in a null resource called null_resource.run_script and realize that you need to rerun the script.
Which of the following commands would you use first?

A. terraform taint null_resource.run_script
B. terraform apply -target=null_resource.run_script
C. terraform validate null_resource.run_script
D. terraform plan -target=null_resource.run_script

**Answer: A**

**Explanation:**

The correct answer is **A. terraform taint null_resource.run_script**.

Here's why: Terraform uses a state file to track the resources it manages. When a resource, including a null_resource, is created or modified, its state is recorded. By default, Terraform will not re-execute provisioners unless it detects a change in the resource configuration. To force a provisioner to run again, even if the configuration hasn't changed, you need to "taint" the resource. Tainting tells Terraform that the resource is somehow out of sync with the desired state, thus requiring a refresh during the next apply. terraform taint null_resource.run_script marks the null_resource.run_script resource as tainted. The next terraform apply will then recreate the null resource, triggering the local-exec provisioner to execute again. Options B, C and D, are incorrect for different reasons. terraform apply -target=null_resource.run_script would apply only the targeted resource, but without tainting, it would typically not re-run the provisioner if there were no configuration changes. terraform validate null_resource.run_script is used for validating the configuration file only, not for triggering resource actions. terraform plan -target=null_resource.run_script would create a plan only for the resource, it doesn't actually trigger the resource creation or provisioner execution. It will, however, show you what changes will occur if you apply that plan.

**Authoritative Links for further research:**

**Terraform taint command:** https://www.terraform.io/cli/commands/taint
**Terraform null_resource:** https://registry.terraform.io/providers/hashicorp/null/latest/docs/resources/resource
**Terraform provisioners:** https://www.terraform.io/language/resources/provisioners/syntax

**Question: 7** CertyIQ

Which provisioner invokes a process on the resource created by Terraform?

A. remote-exec
B. null-exec
C. local-exec
D. file

**Answer: A**

**Explanation:**

The correct answer is A, remote-exec. Terraform provisioners are mechanisms used to perform actions on resources after they are created. remote-exec is designed to execute commands on a remote machine, typically the resource that Terraform has just provisioned. This process is crucial for configuring software, deploying applications, or performing other post-creation tasks directly on the target environment. It establishes an SSH connection to the remote server (or other appropriate protocol) and runs the specified script or command. Option B, null-exec, doesn't interact with a resource; it's useful for triggering actions without modifying an actual resource. Option C, local-exec, executes commands on the machine where Terraform is being run, not the created resource itself. Option D, file, is for copying files to a remote machine, not for executing processes. Therefore, remote-exec is the only provisioner listed that specifically invokes a process on the newly created resource. This action is vital for fully setting up and configuring a cloud infrastructure. Further exploration into Terraform provisioners can be done at the official documentation: https://developer.hashicorp.com/terraform/language/resources/provisioners/syntax and https://developer.hashicorp.com/terraform/language/resources/provisioners/remote-exec.

**Question: 8** CertyIQ

Which of the following is not true of Terraform providers?

A. Providers can be written by individuals
B. Providers can be maintained by a community of users
C. Some providers are maintained by HashiCorp
D. Major cloud vendors and non-cloud vendors can write, maintain, or collaborate on Terraform providers
E. None of the above

**Answer: E**

**Explanation:**

The correct answer is E, "None of the above," because all the statements A through D are true regarding Terraform providers. Terraform providers are plugins that allow Terraform to interact with various infrastructure platforms, APIs, and services. A provider's core functionality is to translate Terraform configuration into API calls for specific services. Option A is valid, as individuals can develop and publish Terraform providers to extend the scope of Terraform. Option B is also true, community-driven provider maintenance ensures the long-term viability and evolution of less commercially focused or niche service integrations. Option C is factual, because HashiCorp actively develops and maintains several prominent providers, such as those for AWS, Azure, and GCP. Statement D is also accurate, large cloud vendors and other companies are critical in building and maintaining providers for their respective APIs and products. Therefore, since all statements are correct, "None of the above" is the only suitable answer, which highlights that all the previous options describe the nature of Terraform providers accurately.

Authoritative links for further research:

**Terraform Providers Documentation:** https://www.terraform.io/docs/providers/index.html
**HashiCorp's Guide to Writing Providers:** https://www.terraform.io/docs/providers/guides/plugin-development.html

**Question: 9** CertyIQ

What command does Terraform require the first time you run it within a configuration directory?

A. terraform import
B. terraform init
C. terraform plan
D. terraform workspace

**Answer: B**

**Explanation:**

The correct answer is **B. terraform init**. Terraform, before executing any operation like planning or applying configurations, needs to initialize its working directory. This initialization process, triggered by terraform init, is crucial for setting up the necessary plugins and modules. It downloads the required provider plugins (like AWS, Azure, GCP) which allow Terraform to interact with the specified infrastructure. Furthermore, it initializes the backend state, where Terraform keeps track of the current infrastructure it manages. If the configuration uses external modules, terraform init retrieves those as well. Without this step, Terraform won't know how to provision the specified resources or where to store the state information. Subsequent commands like terraform plan and terraform apply depend on a successful terraform init. The other options are incorrect; terraform import is used to bring existing infrastructure under Terraform management, terraform plan creates an execution plan, and terraform workspace allows managing different environments. Therefore, terraform init is the indispensable first command in a new Terraform directory.

For further research, refer to the official Terraform documentation:

**Initialize a Working Directory:** https://developer.hashicorp.com/terraform/cli/commands/init
**Terraform Workflow:** https://developer.hashicorp.com/terraform/tutorials/aws-get-started/terraform-workflow

**Question: 10** CertyIQ

You have deployed a new webapp with a public IP address on a cloud provider. However, you did not create any outputs for your code.
What is the best method to quickly find the IP address of the resource you deployed?

A. Run terraform output ip_address to view the result
B. In a new folder, use the terraform_remote_state data source to load in the state file, then write an output for each resource that you find the state file
C. Run terraform state list to find the name of the resource, then terraform state show to find the attributes including public IP address
D. Run terraform destroy then terraform apply and look for the IP address in stdout

**Answer: C**

**Explanation:**

The correct answer is **C: Run terraform state list to find the name of the resource, then terraform state show to find the attributes including the public IP address.** Here's why:

Terraform maintains a state file that tracks the resources it manages. This state file holds crucial information, including resource names and their attributes (like the public IP address). When a resource is created without specific output definitions, you can still retrieve its details from the state file.

Option A is incorrect because terraform output ip_address would only work if you had previously defined an output named "ip_address" in your Terraform configuration.

Option B is overly complex and inefficient. Using terraform_remote_state is generally used to access states from other Terraform configurations and is not necessary for this case. It also requires setting up another configuration file. It involves unnecessary overhead for a simple retrieval of attributes of the current state.

Option D is inappropriate since running terraform destroy and terraform apply to find the IP address is a destructive approach. It would destroy and re-create the infrastructure, which can be disruptive and unnecessary, just to find the IP.

terraform state list will show you all the resources managed by terraform in its state. After identifying the resource name, terraform state show <resource_name> allows you to view the detailed attributes of a specific resource, including the public IP address. This is the most direct and non-destructive way to extract the required information when outputs weren't defined. The state is the single source of truth about your managed infrastructure, and leveraging it directly is the correct approach for this problem.

**Authoritative Links:**

**Terraform State:** https://www.terraform.io/language/state
**Terraform State Commands:** https://www.terraform.io/cli/commands/state

**Question: 11** CertyIQ

Which of the following is not a key principle of infrastructure as code?

A. Versioned infrastructure
B. Golden images
C. Idempotence
D. Self-describing infrastructure

**Answer: B**

**Explanation:**

The correct answer is B, Golden images, as it is not a key principle of Infrastructure as Code (IaC). IaC focuses on managing and provisioning infrastructure through code rather than manual processes. Versioned infrastructure (A) is crucial as it allows tracking changes and reverting to previous states, providing auditability and rollbacks. Idempotence (C) is vital; applying the same IaC configuration multiple times should result in the same final state, preventing unintended changes or errors. Self-describing infrastructure (D) means the code itself provides comprehensive information about the infrastructure's current state, enhancing maintainability and understanding. Golden images, on the other hand, are pre-configured virtual machine templates or snapshots used for rapid deployment. While they can be used in conjunction with IaC, they don't form a core principle of how IaC operates. Golden images are more aligned with a deployment strategy than the fundamental principles of managing infrastructure through code. IaC is about automating the creation and management of infrastructure, while golden images focus on the deployment of already-created components. Think of IaC as the recipe and golden images as pre-made meal kits. Though they both aim for similar goals, they are distinct approaches. Therefore, while golden images are valuable in many deployments, they are not a key principle defining IaC itself.

**Supporting Links:**

**HashiCorp's Infrastructure as Code:** https://www.hashicorp.com/resources/what-is-infrastructure-as-code
**AWS - What is Infrastructure as Code?** https://aws.amazon.com/what-is/infrastructure-as-code/

**Microsoft Azure - What is Infrastructure as Code?** https://azure.microsoft.com/en-us/resources/cloud-computing-dictionary/what-is-infrastructure-as-code

**Question: 12** CertyIQ

Terraform variables and outputs that set the "description" argument will store that description in the state file.

A. True
B. False

**Answer: B**

**Explanation:**

The statement that Terraform variable and output descriptions are stored in the state file is incorrect. Terraform's state file primarily tracks the infrastructure resources managed by Terraform, their configurations, and their unique identifiers. It's a critical mapping between your configuration and the real-world infrastructure. Variable and output descriptions are metadata intended for human readability and documentation purposes within the Terraform code. They do not influence the management of resources, therefore are not needed for Terraform's core functionality regarding the resources lifecycle. Instead, these descriptions are parsed and displayed by Terraform when running commands like terraform plan or terraform apply and also appear in the documentation generated by tools like terraform-docs. This ensures that users understand the purpose and intended usage of variables and outputs without cluttering the state file with non-essential metadata. Storing these descriptions in the state would unnecessarily increase its size and complexity, potentially impacting performance. The state file needs to be efficient in tracking the current infrastructure state. Furthermore, description can easily be retrieved from the Terraform configuration files when required and do not need to be duplicated in the state file. The focus of the state file is solely on resource tracking and management, keeping it concise and effective.

https://www.terraform.io/language/statehttps://www.terraform.io/language/values/variableshttps://www.terraform

**Question: 13** CertyIQ

What is the provider for this fictitious resource?

```
resource "aws_vpc" "main" {
    name = "test"
}
```

A. vpc
B. main
C. aws
D. test

**Answer: C**

**Explanation:**

Reference:
https://docs.aws.amazon.com/cloudformation-cli/latest/userguide/resource-types.html

**Question: 14** CertyIQ

If you manually destroy infrastructure, what is the best practice reflecting this change in Terraform?

A. Run terraform refresh
B. It will happen automatically
C. Manually update the state fire
D. Run terraform import

**Answer: A**

**Explanation:**

The correct answer is **A. Run terraform refresh**. Here's why: Terraform's state file is the source of truth about the infrastructure it manages. When infrastructure is manually destroyed outside of Terraform, the state file becomes inconsistent with reality. The terraform refresh command reconciles this discrepancy by querying the actual infrastructure and updating the state file to reflect the current status. It does not modify the infrastructure itself; it only updates Terraform's knowledge of it. Option B is incorrect because Terraform does not automatically detect out-of-band changes. Option C is not recommended as manually altering the state file can lead to corruption and future management issues. Option D, terraform import, is used to bring existing resources under Terraform management, not to reflect manual destruction. Running terraform refresh is the crucial step to ensure future Terraform actions are consistent with the real world and avoid errors. Without refreshing, Terraform might attempt to modify or delete resources that no longer exist, or fail to acknowledge that something is already gone. Therefore, using terraform refresh is the correct way to handle infrastructure changes made outside of Terraform's control. This guarantees Terraform's understanding of the deployed resources matches reality.

**Authoritative Links:**

**Terraform refresh:** https://www.terraform.io/cli/commands/refresh
**Terraform State:** https://www.terraform.io/language/state

**Question: 15** CertyIQ

What is not processed when running a terraform refresh?

A. State file
B. Configuration file
C. Credentials
D. Cloud provider

**Answer: B**

**Explanation:**

The correct answer is B, the configuration file, because terraform refresh primarily focuses on synchronizing Terraform's state file with the current state of the infrastructure. Let's break down why.

Terraform maintains a "state" file that acts as a mapping between resources defined in your configuration files and the actual infrastructure resources in your cloud environment. When you run terraform refresh, Terraform queries your cloud provider's APIs to understand the current status of the managed resources. It then updates the state file to reflect these real-world changes. This ensures that the state file accurately

represents the current infrastructure. It doesn't, however, modify or re-evaluate the configuration files themselves.

The configuration file (written in HCL) defines the desired state of your infrastructure. It's the blueprint for what you want to build or manage. terraform refresh does not analyze the configuration to change its behavior, and no evaluation of the .tf files takes place. Instead, it reads its existing state file, goes to the Cloud Provider, pulls the current state, and updates the state file with any detected differences. Credentials for the cloud provider are necessary for API interactions to query infrastructure, and they are used during a refresh. The state file itself is the target of this process; it's the data that is being updated, which means it is processed.

In summary, terraform refresh is solely about ensuring the state file accurately mirrors the reality of your deployed infrastructure. It is not a tool for updating configuration files or how your infrastructure is defined.

For further information, consult these authoritative links:

1. Terraform documentation on refresh: https://www.terraform.io/cli/commands/refresh
2. Terraform documentation on state: https://www.terraform.io/language/state

**Question: 16** CertyIQ

What information does the public Terraform Module Registry automatically expose about published modules?

A. Required input variables
B. Optional inputs variables and default values
C. Outputs
D. All of the above
E. None of the above

**Answer: D**

**Explanation:**

The correct answer is indeed D, "All of the above." The public Terraform Module Registry is designed to facilitate module discovery and reuse. To achieve this, it automatically exposes comprehensive metadata about published modules, making it transparent and predictable for users. This includes required input variables, which are essential for the module to function correctly. Without this information, users wouldn't know what data they need to provide. Similarly, the registry exposes optional input variables, alongside their default values, giving users flexibility and control over module behavior while allowing them to leverage reasonable defaults if desired. Finally, the registry provides information about the module's outputs, enabling users to integrate the module's results into their broader infrastructure configuration. These outputs are the data the module provides once its resources are created and are crucial for chaining modules together or accessing resource attributes. By revealing all of these aspects – required inputs, optional inputs with defaults, and outputs – the registry allows for an efficient and understandable module-based infrastructure as code workflow. Without this level of detail, users would face significant challenges in utilizing the published modules effectively. The exposed information ensures transparency and enables confident module consumption within the Terraform ecosystem.

Relevant documentation can be found on the official HashiCorp website:

**Terraform Module Registry Documentation:** https://www.terraform.io/docs/registry/index.html
**Publishing Modules:** https://www.terraform.io/docs/registry/modules/publish.html

**Question: 17** CertyIQ

If a module uses a local values, you can expose that value with a terraform output.

A. True
B. False

**Answer: A**

**Explanation:**

The statement "If a module uses a local values, you can expose that value with a terraform output" is indeed **True**. Local values in Terraform modules are intended for internal calculations and data manipulation within the module's scope. However, to make these values accessible to the calling configuration (the root module or another module that uses this module), they need to be explicitly exposed using output values. An output value serves as an interface through which a module communicates selected internal values to its external environment. This mechanism allows for modularity and reusability by enabling the passing of calculated information or critical parameters derived within the module to the caller. Without using outputs, local values remain confined within the module's boundaries. This mechanism prevents direct access and modifications from outside the module, reinforcing encapsulation and good practice in infrastructure-as-code principles. Therefore, the correct answer is **A. True**.

Further reading on Terraform modules, local values, and output values can be found at these official HashiCorp Terraform documentation links:

1. **Modules:** https://developer.hashicorp.com/terraform/language/modules
2. **Local Values:** https://developer.hashicorp.com/terraform/language/values/locals
3. **Output Values:** https://developer.hashicorp.com/terraform/language/values/outputs

**Question: 18** CertyIQ

You should store secret data in the same version control repository as your Terraform configuration.

A. True
B. False

**Answer: B**

**Explanation:**

Storing secret data directly within the same version control repository as your Terraform configuration is a highly discouraged practice due to significant security risks. Version control systems like Git are designed to track changes to code and configurations; they are not inherently secure for storing sensitive information such as API keys, passwords, or database credentials. If secrets are committed to the repository, they become part of the commit history, potentially exposing them to anyone with access to the repository, even after the secrets are removed from the current configuration. This broad exposure increases the likelihood of unauthorized access, data breaches, and compromises. Instead, secure secret management solutions are essential. These tools encrypt secrets at rest and control access to them based on defined policies. Popular options include HashiCorp Vault, AWS Secrets Manager, Azure Key Vault, and Google Cloud Secret Manager. These tools provide a centralized, auditable, and more secure approach to handling secrets required by infrastructure as code. Terraform integrates with these solutions to dynamically fetch secrets during runtime, keeping them out of the code and secure. Therefore, the correct answer is **False**.Here are some links for further research:

**HashiCorp Vault:** https://www.vaultproject.io/
**Terraform Secret Management:** https://developer.hashicorp.com/terraform/tutorials/configuration/secrets
**AWS Secrets Manager:** https://aws.amazon.com/secrets-manager/
**Azure Key Vault:** https://azure.microsoft.com/en-us/products/key-vault/
**Google Cloud Secret Manager:** https://cloud.google.com/secret-manager

**Question: 19** CertyIQ

Which of the following is not a valid string function in Terraform?

A. split
B. join
C. slice
D. chomp

**Answer: C**

**Explanation:**

The correct answer is C, slice, because it is not a built-in string function in Terraform. Terraform provides a variety of built-in functions for string manipulation, which are essential when working with dynamic infrastructure configurations. These functions allow users to process and format text data, making configurations more flexible and manageable. split, as in option A, is a valid function that divides a single string into a list of strings based on a delimiter. join, option B, performs the opposite, concatenating elements of a list into a single string using a specified delimiter. chomp, option D, removes trailing newline characters from a string. However, slice isn't a string-specific function in Terraform; instead, it's used with collections (lists and maps) for extracting a subsection of elements, not manipulating string characters. To manipulate substrings, other string functions must be used. Therefore, within the context of string functions, slice is the outlier. For authoritative information on Terraform's string functions, refer to the official Terraform documentation: https://www.terraform.io/language/functions/strings and https://www.terraform.io/language/functions/collection. These sources clearly outline the available string functions and the behavior of the slice function with collections.

**Question: 20** CertyIQ

You have provisioned some virtual machines (VMs) on Google Cloud Platform (GCP) using the gcloud command line tool. However, you are standardizing with
Terraform and want to manage these VMs using Terraform instead.
What are the two things you must do to achieve this? (Choose two.)

A. Provision new VMs using Terraform with the same VM names
B. Use the terraform import command for the existing VMs
C. Write Terraform configuration for the existing VMs
D. Run the terraform import-gcp command

**Answer: BC**

**Explanation:**

The correct answer is **B and C**. To bring pre-existing GCP VMs under Terraform management, you cannot simply create new VMs with the same names (option A). This would result in conflicting resources and is not

the intended way to adopt existing infrastructure. Option D, "terraform import-gcp," is not a valid command. While terraform import is the correct mechanism, there is no specific import-gcp subcommand.

Therefore, the correct approach involves two key steps. Firstly, you must write the Terraform configuration files (option C) describing the existing GCP VMs. This includes specifying the relevant attributes of each VM such as the machine type, image, network configuration, etc. These configuration files will serve as the desired state in Terraform. Secondly, you must use the terraform import command (option B) to map the resources defined in your Terraform configuration to their corresponding existing instances in GCP. The import command establishes the link between your Terraform state and the physical resources in the cloud. This does not alter the existing GCP VMs, but allows Terraform to track them and manage future changes through its configuration and state.

In essence, Terraform needs to know what already exists and needs to have a record (state) of these resources. Importing enables Terraform to gain that knowledge from your existing infrastructure. Attempting to manage resources without importing them will result in inconsistencies and can lead to resources being recreated, causing disruption. This two-step process effectively transitions the management of existing GCP VMs from gcloud to Terraform.

Relevant Links:

Terraform Import documentation: https://developer.hashicorp.com/terraform/cli/import
Google Cloud Platform provider for Terraform:
https://registry.terraform.io/providers/hashicorp/google/latest/docs

**Question: 21** CertyIQ

You have recently started a new job at a retailer as an engineer. As part of this new role, you have been tasked with evaluating multiple outages that occurred during peak shopping time during the holiday season. Your investigation found that the team is manually deploying new compute instances and configuring each compute instance manually. This has led to inconsistent configuration between each compute instance.
How would you solve this using infrastructure as code?

A. Implement a ticketing workflow that makes engineers submit a ticket before manually provisioning and configuring a resource

B. Implement a checklist that engineers can follow when configuring compute instances

C. Replace the compute instance type with a larger version to reduce the number of required deployments

D. Implement a provisioning pipeline that deploys infrastructure configurations committed to your version control system following code reviews

**Answer: D**

**Explanation:**

The most effective solution to inconsistent compute instance configurations and manual deployments is **D. Implement a provisioning pipeline that deploys infrastructure configurations committed to your version control system following code reviews.** This approach leverages the principles of Infrastructure as Code (IaC). IaC treats infrastructure definitions as code, allowing them to be version-controlled, reviewed, and automated. By storing configurations in a version control system like Git, teams gain a single source of truth for their infrastructure. Changes are tracked, and rollbacks are simplified. Code reviews ensure that configurations meet the team's standards and prevent errors. The provisioning pipeline automates the deployment of these configurations. This automation eliminates manual steps, reducing human errors and ensuring consistency. Options A and B address the manual process but don't inherently solve the consistency problem. Option C, while a valid scaling consideration, does not directly address the inconsistent configurations. The core issue is a lack of repeatable and automated infrastructure management, which IaC directly addresses. A provisioning pipeline with IaC is the best practice solution for reliably and repeatedly

deploying infrastructure with consistent configurations. This approach leads to faster deployments, fewer errors, and improved infrastructure management.

**Authoritative Links:**

**Infrastructure as Code (IaC):** https://www.redhat.com/en/topics/automation/what-is-infrastructure-as-code-iac
**Git Version Control:** https://git-scm.com/
**CI/CD Pipelines:** https://www.atlassian.com/continuous-delivery/principles/continuous-integration-vs-continuous-delivery

**Question: 22** CertyIQ

terraform init initializes a sample main.tf file in the current directory.

A. True
B. False

**Answer: B**

**Explanation:**

The statement that terraform init initializes a sample main.tf file is incorrect. terraform init's primary function is to initialize a Terraform working directory. This includes downloading necessary provider plugins and setting up the backend configuration based on the configuration files already present in the directory. It doesn't create or modify any .tf files, including main.tf. If no main.tf file (or other configuration files) exists, terraform init will still complete successfully, setting up the working environment, but it won't fabricate a configuration. The user is expected to create configuration files containing resource definitions. The purpose of terraform init is to prepare the environment to operate on existing configurations, not to create them. A main.tf file would typically contain resource blocks, providers, and other configuration specifics that define infrastructure-as-code. Terraform's principle is to declaratively define the desired state through the HCL syntax in files like main.tf. Therefore, users are responsible for crafting the desired configuration, not terraform init.

For further information, you can consult the official Terraform documentation:

Terraform init Documentation: This page provides a comprehensive overview of the terraform init command, its purpose, and its functionality.
Terraform Configuration Syntax: This section covers the syntax of Terraform configuration files, clarifying that users define the configuration themselves.
Terraform Workflow: This tutorial outlines the typical workflow, emphasizing that you write configuration files before you initiate the working directory with init.

**Question: 23** CertyIQ

Which two steps are required to provision new infrastructure in the Terraform workflow? (Choose two.)

A. Destroy
B. Apply
C. Import
D. Init
E. Validate

**Answer: BD**

**Explanation:**

The correct answer, **B. Apply** and **D. Init**, represents the two core steps needed for provisioning infrastructure with Terraform. **Init** is the first step and it initializes the working directory. This involves downloading necessary provider plugins and modules that Terraform needs to interact with specific infrastructure platforms (like AWS, Azure, or GCP). Without a successful terraform init, Terraform cannot interpret the configuration files or interact with the required APIs. This prepares Terraform's environment to understand and manage the specified resources.

Once the environment is initialized, the **Apply** command then takes the configuration defined in Terraform files and executes it. It creates or modifies the infrastructure as described in the code. This involves making API calls to the providers, which then communicate with the underlying cloud services to provision the requested resources. Apply compares the desired state (defined in the configuration files) with the current state, and then implements the necessary changes to achieve the desired state.

Destroy, Import, and Validate represent other operations within Terraform. Destroy is used to remove existing resources, Import is used to manage existing infrastructure not initially created with Terraform, and Validate checks the syntax of the configuration but does not affect resource provisioning. Therefore, Apply and Init are the essential steps for actual provisioning.

**Further Research:**

**Terraform Workflow:** https://developer.hashicorp.com/terraform/tutorials/aws-get-started/terraform-workflow
**Terraform init:** https://developer.hashicorp.com/terraform/cli/commands/init
**Terraform apply:** https://developer.hashicorp.com/terraform/cli/commands/apply

**Question: 24** CertyIQ

Why would you use the terraform taint command?

A. When you want to force Terraform to destroy a resource on the next apply
B. When you want to force Terraform to destroy and recreate a resource on the next apply
C. When you want Terraform to ignore a resource on the next apply
D. When you want Terraform to destroy all the infrastructure in your workspace

**Answer: B**

**Explanation:**

The correct answer is **B. When you want to force Terraform to destroy and recreate a resource on the next apply**. The terraform taint command is specifically used to mark a resource as "tainted" within Terraform's state file. This action signals to Terraform that the resource is no longer in a desired state and should be replaced. When Terraform performs a subsequent apply, it will plan to destroy the tainted resource and then recreate it, effectively forcing a replacement. This is useful in situations where a resource has become corrupted, misconfigured, or needs a refresh for any other reason, but modifying the configuration is not sufficient. It differs from destroying all infrastructure, which is done with terraform destroy, and ignoring resources, which is done with other configuration techniques. Tainting only targets a specific resource, not all of them. The terraform taint command doesn't simply "destroy," it triggers a destroy followed by a create to refresh the infrastructure based on the code.

For more information and in-depth documentation, please refer to the official HashiCorp Terraform

documentation:

Terraform Taint Command
Understanding Terraform State

**Question: 25** CertyIQ

Terraform requires the Go runtime as a prerequisite for installation.

A. True
B. False

**Answer: B**

**Explanation:**

The assertion that Terraform requires the Go runtime as a prerequisite is incorrect. Terraform is indeed written in Go, but its executable is distributed as a pre-compiled binary. Users don't need to have the Go development environment installed on their systems to run Terraform. The compiled binary packages all the necessary dependencies within it, allowing for direct execution without needing a separate Go installation. This simplifies the user experience, as it removes the overhead of managing a specific programming language runtime just to use the tool. The independence from a Go runtime makes Terraform portable across different operating systems where the binaries are supported. The focus is on providing a seamless and easy way to manage infrastructure, and dependency on Go would contradict this objective. Instead, users download and run the appropriate binary for their system from HashiCorp's official website. This packaged distribution is a core aspect of Terraform's design for ease of use and wide adoption. Refer to the official Terraform documentation for details on installation: https://www.terraform.io/downloads. HashiCorp's focus is user-friendly tooling, and requiring a Go runtime for general use would not align with this goal.

**Question: 26** CertyIQ

When should you use the force-unlock command?

A. You see a status message that you cannot acquire the lock
B. You have a high priority change
C. Automatic unlocking failed
D. You apply failed due to a state lock

**Answer: C**

**Explanation:**

The correct answer is **C. Automatic unlocking failed**. The force-unlock command in HashiCorp tools like Terraform is a last-resort option for situations where a state lock cannot be automatically released. Typically, a state lock is acquired during operations like apply to prevent concurrent modifications that could corrupt the state file. If a process acquiring the lock fails or is interrupted before releasing it, the lock can remain indefinitely, blocking subsequent operations. Automatic unlocking mechanisms should release the lock under normal circumstances; when they fail to do so, manual intervention using force-unlock becomes necessary.

Option A, "You see a status message that you cannot acquire the lock," indicates a normal locking situation, not one requiring force-unlock. This is expected behavior when another operation holds the lock. Option B, "You have a high priority change," doesn't justify force-unlocking; a high-priority change still needs to respect the

locking mechanism to ensure data integrity. Option D, "You apply failed due to a state lock," highlights that the apply failed due to a lock, which should trigger automatic unlocking attempts first, not immediately force-unlocking it. Force unlocking disrupts safe workflows and should only be considered when other unlock attempts have failed. Misuse of force-unlock can result in state corruption or inconsistencies if multiple operations are performed without proper coordination.

Therefore, the force-unlock command is only appropriate when the system itself fails to release a lock automatically, often due to process crashes or other unforeseen situations leaving behind a stale lock. This requires careful consideration, awareness of the risk of state corruption, and should follow other troubleshooting steps.

Authoritative links for further research:

**Terraform documentation on state locking:** https://www.terraform.io/language/state/locking
**Terraform force-unlock command:** https://www.terraform.io/cli/commands/force-unlock

**Question: 27** CertyIQ

Terraform can import modules from a number of sources `" which of the following is not a valid source?

A. FTP server
B. GitHub repository
C. Local path
D. Terraform Module Registry

**Answer: A**

**Explanation:**

The correct answer is A. FTP server. Terraform, a popular Infrastructure as Code (IaC) tool, is designed to fetch modules from specified sources to enhance code reusability and consistency. It natively supports several locations for module retrieval, including GitHub repositories (B), allowing direct access to version-controlled modules stored in remote repositories. Local path (C) serves as a straightforward source, enabling referencing modules stored on the same system where the Terraform configuration resides. Furthermore, the Terraform Module Registry (D) provides a central hub for publishing and consuming publicly available modules, fostering community collaboration and accelerating development. However, Terraform does not offer direct support for importing modules from an FTP server (A). This is because FTP is a file transfer protocol primarily designed for general-purpose file sharing, not module management with version control or dependency resolution, which are core features of Terraform's module system. Therefore, FTP lacks the required mechanisms for reliable and consistent module importing within the Terraform ecosystem. You can delve deeper into Terraform module sources at the official documentation:
https://developer.hashicorp.com/terraform/language/modules/sources

**Question: 28** CertyIQ

Which of the following is available only in Terraform Enterprise or Cloud workspaces and not in Terraform CLI?

A. Secure variable storage
B. Support for multiple cloud providers
C. Dry runs with terraform plan
D. Using the workspace as a data source

**Answer: A**

**Explanation:**

The correct answer is **A. Secure variable storage**. Here's why:

Terraform, by itself, relies on environment variables, command-line arguments, or terraform.tfvars files to manage variables, which may not be secure for sensitive information. Terraform Enterprise and Terraform Cloud, however, offer secure variable storage solutions through features like encrypted variables at rest and in transit. These variables can be designated as sensitive, preventing them from being printed in logs or displayed in the user interface, ensuring enhanced security and compliance. They typically use their own dedicated backend to secure variables. Terraform CLI does not have a secure mechanism for variable storage built into it, relying instead on external secure storage for secrets. Options like using vaults or secret managers are required when the Terraform CLI is used.

**B. Support for multiple cloud providers** is a core feature of Terraform CLI itself, enabling it to manage resources across various cloud providers (AWS, Azure, GCP, etc.). This is a fundamental aspect of Terraform's design and is available to all its versions, including the CLI. **C. Dry runs with terraform plan** is an essential feature of Terraform CLI, enabling users to preview changes before applying them to infrastructure. It's a key part of the iterative development process. **D. Using the workspace as a data source** is available in both Terraform CLI, utilizing the terraform_remote_state datasource, and in Terraform Cloud/Enterprise, utilizing both terraform_remote_state and other ways of accessing workspace data. The fundamental capability is built into the core Terraform language.

Therefore, the enhanced variable security provided by Terraform Enterprise and Terraform Cloud workspaces, specifically its built-in encrypted secure storage, is not a feature present in the basic Terraform CLI. This difference highlights a key distinction in the enterprise-level security offerings of Terraform's hosted platforms compared to the fundamental capabilities of the CLI.

For further reading:

**Terraform Cloud Variables:** https://developer.hashicorp.com/terraform/cloud/workspaces/variables
**Terraform Cloud Secure Variables:** https://developer.hashicorp.com/terraform/cloud/workspaces/variables#sensitive-variables
**Terraform CLI Variables:** https://developer.hashicorp.com/terraform/language/values/variables
**Terraform CLI terraform_remote_state datasource:** https://registry.terraform.io/providers/hashicorp/terraform/latest/docs/data-sources/remote_state

**Question: 29** CertyIQ

terraform validate validates the syntax of Terraform files.

A. True
B. False

**Answer: A**

**Explanation:**

The statement is indeed true; terraform validate is the command in Terraform used to verify the syntax and internal consistency of your Terraform configuration files. It checks for errors in the HCL (HashiCorp Configuration Language) syntax, ensuring that the code adheres to the rules defined by Terraform and the provider plugins. This command does not interact with any infrastructure or cloud providers; it strictly focuses on the static analysis of the configuration. It confirms if all required arguments are present, that variable

types match the expectations, and that modules are correctly referenced. Successful validation means the syntax is correct and the Terraform files can be processed by the subsequent terraform plan and terraform apply steps. Failing validation indicates that the configuration needs adjustments to comply with Terraform's specifications before deploying any infrastructure. Essentially, terraform validate is a critical first step in a robust infrastructure-as-code workflow, helping to catch errors early in the development cycle. This proactive approach prevents runtime errors and ensures smoother infrastructure deployments.

https://www.terraform.io/cli/commands/validate

**Question: 30** CertyIQ

You have used Terraform to create an ephemeral development environment in the cloud and are now ready to destroy all the infrastructure described by your
Terraform configuration. To be safe, you would like to first see all the infrastructure that will be deleted by Terraform.
Which command should you use to show all of the resources that will be deleted? (Choose two.)

A. Run terraform plan -destroy.

B. This is not possible. You can only show resources that will be created.

C. Run terraform state rm *.

D. Run terraform destroy and it will first output all the resources that will be deleted before prompting for approval.

**Answer: AD**

**Explanation:**

The correct commands to preview resource deletion in Terraform are terraform plan -destroy and terraform destroy.

terraform plan -destroy specifically generates an execution plan that outlines the actions Terraform will take to destroy the infrastructure defined in your configuration. This includes a detailed list of resources that will be deleted, modified (though mostly applicable to non-destroy operations) or no changes will be made. It's a critical command for ensuring you understand the impact of your destroy operation before executing it, promoting safety and reducing the risk of unintended data loss or infrastructure damage. You can research more about terraform plan on the official Terraform documentation: https://www.terraform.io/cli/commands/plan.

terraform destroy, on the other hand, initiates the actual destruction process. However, before taking any actions, it also displays a plan, similar to the output of terraform plan -destroy, which lists the resources to be removed. This provides a second opportunity for review and confirmation before proceeding. Following this display, Terraform prompts for user approval to continue with the destruction. This two-step process provides safety when managing infrastructure. You can learn more about terraform destroy on: https://www.terraform.io/cli/commands/destroy.

Option C, terraform state rm , is incorrect because it removes resources from the Terraform state file without actually destroying the cloud resources themselves. This action only misaligns the state file with the current real-world situation. Also *terraform state rm* requires a specific resource addresses not a . Option B is incorrect since Terraform's planning stage allows you to review changes before they happen, including destructions.

Therefore, options A and D are the correct answers as they both present the resource deletion actions before Terraform modifies any cloud infrastructure.

**Question: 31** CertyIQ

Which of the following is the correct way to pass the value in the variable num_servers into a module with the input servers?

A. servers = num_servers
B. servers = variable.num_servers
C. servers = var(num_servers)
D. servers = var.num_servers

**Answer: D**

**Explanation:**

The correct way to pass the value of a variable into a module in HashiCorp Terraform is by using the var keyword followed by a dot and the variable name. In this scenario, the intention is to pass the value stored in the variable num_servers to the servers input variable of a module. Option D, servers = var.num_servers, achieves this accurately. Terraform uses the var keyword to indicate that you are referencing a variable defined elsewhere in your configuration. The dot notation . allows you to access a specific variable by its name within the var context.

Option A, servers = num_servers, is incorrect because it directly assigns the variable name num_servers as a string, instead of its value. Option B, servers = variable.num_servers, is also incorrect; there isn't a variable keyword to refer to declared variables, it is always var. Option C, servers = var(num_servers), uses the correct var keyword but the incorrect parentheses syntax.

Terraform input variables provide a mechanism to customize modules, allowing them to be reusable with different values based on the user's requirement. When you call a module, using the var attribute with dot notation for module input is crucial for passing actual values dynamically and avoiding hardcoding information. Understanding this practice is fundamental to effectively managing and organizing infrastructure as code with Terraform. Terraform Variables Documentation and Terraform Modules Documentation provide comprehensive resources for learning more about variables and modules.

**Question: 32** CertyIQ

A Terraform provisioner must be nested inside a resource configuration block.

A. True
B. False

**Answer: A**

**Explanation:**

The statement is indeed true. Terraform provisioners, which are used to execute scripts or actions on resources after they've been created, must be defined within the resource configuration block. They aren't standalone entities that can be declared outside the scope of a specific resource. This nesting ensures that the provisioner is directly tied to the resource it modifies or manages. A provisioner operates on an instance, server or service created by a resource, and therefore, it must know what resource to configure or manage. Terraform's configuration syntax mandates that provisioners are declared as a block nested inside a resource block. The provisioning actions defined within the provisioner only execute after the resource has been successfully created. Without this resource nesting, Terraform wouldn't know which resource's lifecycle to

attach the provisioner to. This structural requirement enforces a clear relationship between the resource and the actions that need to be performed on it after its instantiation. It's crucial for Terraform's orchestration model where the dependency chain is managed implicitly via this hierarchy. For instance, if you want to install software on a server, the provisioner (like a remote-exec or file) must be declared inside the resource "aws_instance" block. Attempting to define a provisioner outside a resource block will result in a syntax error. This requirement is fundamental to Terraform's declarative infrastructure-as-code approach.https://developer.hashicorp.com/terraform/language/resources/provisioners/syntax

**Question: 33** CertyIQ

Terraform can run on Windows or Linux, but it requires a Server version of the Windows operating system.

A. True
B. False

**Answer: B**

**Explanation:**

The statement that Terraform requires a server version of Windows is incorrect. Terraform, being a command-line tool, is designed for broad compatibility. It executes on various operating systems, including Windows, macOS, and Linux, and is agnostic to whether the Windows installation is a server or desktop version. It interacts with cloud providers and other infrastructure through APIs, not directly with the underlying OS architecture in a way that necessitates a server edition. The installation packages available on the official HashiCorp website do not differentiate between Windows server and desktop versions. You can download and run Terraform from any compatible Windows system capable of running command line tools. The core functionality of Terraform is centered around infrastructure as code principles and the ability to declaratively define resources irrespective of specific OS-level features provided by a server edition. Therefore, the assertion that a server version is mandated is misleading and a clear misconception about its fundamental design.

Official HashiCorp Terraform Documentation - InstallationOfficial Terraform Downloads Page

**Question: 34** CertyIQ

What does the default "local" Terraform backend store?

A. tfplan files
B. Terraform binary
C. Provider plugins
D. State file

**Answer: D**

**Explanation:**

The correct answer is D, the state file. The "local" backend in Terraform, by default, stores the state file on the same machine where Terraform commands are executed. This state file is crucial for Terraform to track the infrastructure it manages. It maps resources defined in your configuration files to the real-world objects they represent. The state file contains information about the resources' attributes, dependencies, and current configurations, allowing Terraform to understand the existing infrastructure and propose changes

accordingly. It also prevents drift by remembering the expected state of the resources. The state file is a JSON file, generally named terraform.tfstate by default. Without the state file, Terraform cannot manage your infrastructure effectively, as it would be unaware of previous deployments. The local backend doesn't store the Terraform binary (B), which is a standalone executable. Nor does it store tfplan files (A), which are temporary execution plans generated before actual resource changes. Provider plugins (C), while essential for Terraform to interact with different cloud providers and services, are not stored by the local backend either. They are typically downloaded separately when you initialize your project. The local backend is useful for learning and experimentation but is not recommended for team-based environments as it poses challenges for collaboration and state file management.

**Authoritative Links:**

**Terraform Backends:** https://www.terraform.io/docs/language/settings/backends/index.html
**Local Backend:** https://www.terraform.io/docs/language/settings/backends/local.html
**Terraform State:** https://www.terraform.io/docs/language/state/index.html

**Question: 35** CertyIQ

You have multiple team members collaborating on infrastructure as code (IaC) using Terraform, and want to apply formatting standards for readability.
How can you format Terraform HCL (HashiCorp Configuration Language) code according to standard Terraform style convention?

A. Run the terraform fmt command during the code linting phase of your CI/CD process
B. Designate one person in each team to review and format everyone's code
C. Manually apply two spaces indentation and align equal sign "=" characters in every Terraform file (*.tf)
D. Write a shell script to transform Terraform files using tools such as AWK, Python, and sed

**Answer: A**

**Explanation:**

The correct answer is **A. Run the terraform fmt command during the code linting phase of your CI/CD process.** This is the most efficient and scalable method for enforcing consistent formatting in Terraform HCL code across a team. terraform fmt is a built-in Terraform command specifically designed to automatically reformat HCL files according to the official Terraform style guide. Integrating this command into the CI/CD pipeline ensures that all code changes are automatically checked for proper formatting before they are merged or deployed. This removes the burden from individual team members to remember and manually apply specific formatting rules. Using a linting process also provides early feedback to developers about formatting issues, preventing poorly formatted code from reaching production. This approach promotes code consistency and readability, which are crucial for collaboration and maintainability in a team environment. Manually formatting code (option C) is prone to error and not scalable. Designating one person to review code (option B) creates a bottleneck and adds extra work. While scripting (option D) is possible, it duplicates the functionality already available in terraform fmt and is more complex. Automating formatting with terraform fmt within CI/CD is the industry best practice for maintaining a consistent Terraform codebase.

**Authoritative Links:**

**Terraform fmt Documentation:** https://www.terraform.io/cli/commands/fmt
**Terraform Style Conventions:** https://www.terraform.io/language/syntax/configuration (See the 'Style Conventions' section)
**CI/CD Best Practices:** https://www.redhat.com/en/topics/devops/what-is-ci-cd (General CI/CD concepts that supports integrating formatting checks)

**Question: 36** CertyIQ

What value does the Terraform Cloud/Terraform Enterprise private module registry provide over the public Terraform Module Registry?

A. The ability to share modules with public Terraform users and members of Terraform Enterprise Organizations
B. The ability to tag modules by version or release
C. The ability to restrict modules to members of Terraform Cloud or Enterprise organizations
D. The ability to share modules publicly with any user of Terraform

**Answer: C**

**Explanation:**

The correct answer is **C. The ability to restrict modules to members of Terraform Cloud or Enterprise organizations.**

Here's the detailed justification:

The core difference between the public Terraform Module Registry and a private module registry within Terraform Cloud or Enterprise lies in access control. The public registry is designed for open sharing, making modules available to anyone. Conversely, the private registry is intended for internal use within an organization.

Option C accurately reflects this distinction. A private module registry empowers organizations to control which teams and individuals can access and use specific Terraform modules. This promotes security, consistency, and governance by preventing unauthorized deployments or modifications. It's crucial for businesses that have proprietary infrastructure configurations or compliance requirements. It ensures that only those within the designated organization can utilize the published modules.

Option A is incorrect because the public registry already allows sharing with public users, and private registries are not for public sharing. Option B, while true for both registries, doesn't highlight the defining benefit of a private registry: access restriction. Option D describes the public registry, not the private one.

Cloud computing principles of security, governance, and isolation strongly support the need for private registries within organizations to avoid exposing intellectual property or allowing unchecked access to infrastructure code. Organizations using Terraform Cloud or Enterprise gain greater control and security over their module usage through the use of a private registry.

**Authoritative Links:**

**HashiCorp Terraform Registry Documentation:** https://www.terraform.io/docs/registry/index.html (This link provides general information on the Terraform Registry.)
**Terraform Cloud Private Registry Documentation:** https://developer.hashicorp.com/terraform/cloud-docs/registry/private-registry (Specific documentation on Terraform Cloud's private registry.)
**Terraform Enterprise Private Registry Documentation:** (HashiCorp documentation for Terraform Enterprise will describe the same concept)

**Question: 37** CertyIQ

Which task does terraform init not perform?

A. Sources all providers present in the configuration and ensures they are downloaded and available locally
B. Connects to the backend

C. Sources any modules and copies the configuration locally

D. Validates all required variables are present

**Answer: D**

**Explanation:**

The correct answer is D because terraform init focuses on setting up the working environment for Terraform, not on validating variables. The primary functions of terraform init include initializing the backend, which is where Terraform state will be stored (option B). It also downloads the necessary provider plugins (option A) required to interact with different infrastructure platforms (e.g., AWS, Azure, GCP). Additionally, terraform init downloads and configures any modules specified in your configuration (option C), bringing reusable code blocks into your project. However, the validation of required variables happens during the terraform plan or terraform apply stages, not during terraform init. These later stages analyze the configuration and compare it to the existing infrastructure (or lack thereof) and thus need to ascertain whether all necessary variables have been given values. Variable validation is a later phase of Terraform's workflow, ensuring that all configurations are complete before any changes are made. The initialization stage sets up the environment, but it does not perform a full configuration check for complete variable definitions, which are relevant during the operational phases.

**Authoritative Links for further research:**

Terraform init documentation: Official Terraform documentation outlining the functionality of terraform init.
Terraform Variables documentation: Official Terraform documentation on how variables function in Terraform configuration.
Terraform Workflow: Overview of the complete Terraform workflow.

**Question: 38**



You have declared a variable called var.list which is a list of objects that all have an attribute id.
Which options will produce a list of the IDs? (Choose two.)

A. for o in var.list : o => o.id

B. var.list[*].id

C. [ var.list[*].id ]

D. [ for o in var.list : o.id ]

**Answer: BD**

**Explanation:**

Here's a detailed justification for why options B and D are correct for extracting a list of IDs from a list of objects in HashiCorp Configuration Language (HCL), used by tools like Terraform.

Option B, var.list[*].id, employs the splat operator (*[*]*). This operator is fundamental in HCL for traversing all elements within a collection (in this case, a list). It efficiently applies the attribute selection (.id) to each object inside var.list, resulting in a new list comprised solely of those id values.

Option D, [ for o in var.list : o.id ], leverages a for expression within list construction brackets ([]). This is another valid HCL mechanism for transforming lists. The for o in var.list part iterates through each object (named o) in the list var.list. Then, for every o, it extracts its id attribute and adds it to the new list.

Option A, for o in var.list : o => o.id , is incorrect as it attempts to create a map, not a list. The => syntax signifies a key-value association, inappropriate for simple ID extraction. Option C, [ var.list[*].id ], while

syntactically valid, results in a list containing another list instead of a flat list of IDs. It creates a list containing the result of the *var.list[]*.id expression, which is already a list.

The splat operator and for expressions are core features of HCL designed for manipulating complex data structures. Both options B and D demonstrate idiomatic ways to extract specific data from lists of objects. Understanding these concepts allows for the efficient and concise manipulation of configuration data, enhancing the power of declarative infrastructure-as-code tools.

For more in-depth information on HCL constructs, particularly the splat operator and for expressions, consult the official HashiCorp documentation:

**Splat Operator:** https://developer.hashicorp.com/terraform/language/expressions/splat
**For Expressions:** https://developer.hashicorp.com/terraform/language/expressions/for

**Question: 39** CertyIQ

Which argument(s) is (are) required when declaring a Terraform variable?

A. type
B. default
C. description
D. All of the above
E. None of the above

**Answer: E**

**Explanation:**

The correct answer is **E. None of the above**. Terraform variables, when declared, do not require any specific arguments. While type, default, and description are commonly used and highly recommended for clarity and usability, they are all optional. A variable can be declared with just a name, and Terraform will treat it as a string by default. The type argument specifies the data type (string, number, list, map, etc.) if you expect input other than a string. The default argument provides a fallback value if no explicit value is provided for the variable. The description argument provides human-readable context about the purpose of the variable. Not providing a type may lead to unexpected behaviors if you expect a specific data type when working with variables. Similarly, not providing default values and only defining a variable is permissible but, you have to specify a value for it whenever terraform tries to use it. It's critical to define descriptive variables for better readability and maintainability, but these are best practices and not requirements for a variable declaration. You may consider reviewing Terraform's documentation on Variables for more information.

Here are some authoritative links for further research:

**Terraform Variables Documentation:** https://developer.hashicorp.com/terraform/language/values/variables
**Best Practices for Terraform Variables:** https://spacelift.io/blog/terraform-variables-best-practices

**Question: 40** CertyIQ

When using a module block to reference a module stored on the public Terraform Module Registry such as:

```
module "consul" {
  source = "hashicorp/consul/aws"
}
```

How do you specify version 1.0.0?

A. Modules stored on the public Terraform Module Registry do not support versioning
B. Append ?ref=v1.0.0 argument to the source path
C. Add version = "1.0.0" attribute to module block
D. Nothing " modules stored on the public Terraform Module Registry always default to version 1.0.0

**Answer: C**

**Explanation:**

C is correct answer.

The version argument accepts a version constraint string. Terraform will use the newest installed version of the module that meets the constraint; if no acceptable versions are installed, it will download the newest version that meets the constraint.

Version constraints are supported only for modules installed from a module registry, such as the public Terraform Registry or Terraform Cloud's private module registry. Other module sources can provide their own versioning mechanisms within the source string itself, or might not support versions at all. In particular, modules sourced from local file paths do not support version; since they're loaded from the same source repository, they always share the same version as their caller.

Reference:

https://www.terraform.io/language/modules/syntax#version

**Question: 41** CertyIQ

What features does the hosted service Terraform Cloud provide? (Choose two.)

A. Automated infrastructure deployment visualization
B. Automatic backups
C. Remote state storage
D. A web-based user interface (UI)

**Answer: CD**

**Explanation:**

The correct answer is **C. Remote state storage** and **D. A web-based user interface (UI)**. Terraform Cloud, being a managed service, offers core functionalities that simplify infrastructure management.

**Remote state storage (C)** is a fundamental feature, enabling collaborative infrastructure management. Instead of storing Terraform state files locally or in shared drives, which can lead to inconsistencies and data corruption, Terraform Cloud provides a secure and reliable central repository for storing state. This allows teams to work concurrently on the same infrastructure, mitigating state management complexities. It also provides version control of your state file.

**A web-based user interface (UI) (D)** is another essential feature, simplifying Terraform workflows. This interface provides a visual way to manage workspaces, view resource deployments, and track changes. The UI makes infrastructure management more accessible to teams, not only technical individuals, and offers real-time insight into the environment. It acts as the control panel for managing Terraform deployments within the cloud platform.

**Automated infrastructure deployment visualization (A)** is a capability offered by Terraform Cloud, but it is not a fundamental, foundational feature. It is more of an enhancement or an additional feature. While it presents infrastructure in a user-friendly manner, this isn't the core focus.

**Automatic backups (B)** is a feature often associated with infrastructure management tools. However, Terraform Cloud, though it keeps backups of the state data itself, is not directly an infrastructure backup service. Instead, it manages infrastructure configuration. The state file is critical for Terraform's proper operation and is backed up, but it is not the same as backing up the resources that Terraform manages.

Therefore, **remote state storage** and **a web-based user interface** are the two most prominent, core features that Terraform Cloud provides. They are the underlying pillars that allow teams to collaborate on and manage infrastructure through Terraform.

**Further Research:**

**Terraform Cloud Documentation:** https://www.terraform.io/cloud
**HashiCorp Learn - Terraform Cloud:** https://learn.hashicorp.com/terraform/cloud

**Question: 42** CertyIQ

Where does the Terraform local backend store its state?

A. In the /tmp directory
B. In the terraform file
C. In the terraform.tfstate file
D. In the user's terraform.state file

**Answer: C**

**Explanation:**

The correct answer is C, the terraform.tfstate file. The Terraform local backend, by default, stores its state in a file named terraform.tfstate within the working directory where terraform init is executed. This state file is crucial for Terraform to understand the current infrastructure managed by the configuration. It keeps a record of resources created, their attributes, and relationships. When Terraform applies changes, it compares the current state with the desired configuration to determine what needs to be modified. Options A, B, and D are incorrect. The /tmp directory (A) is a temporary location and not used by default for state storage. The terraform file (B) is the configuration file defining the desired infrastructure, not the state. While users can create a file named terraform.state (D) through configuration, the default location and name for local backend state is terraform.tfstate. The terraform.tfstate file is a critical component for Terraform's functionality, enabling it to track, modify, and delete resources according to the declared configuration. Without this state file, Terraform wouldn't be able to manage the infrastructure consistently. The file should be carefully managed to prevent accidental loss or corruption, and it is often recommended to use a remote backend for collaborative environments.

For further research, refer to the official Terraform documentation on backends:

Terraform Backends

Terraform Local Backend

**Question: 43** CertyIQ

Which option can not be used to keep secrets out of Terraform configuration files?

A. A Terraform provider
B. Environment variables
C. A -var flag
D. secure string

**Answer: D**

**Explanation:**

The correct answer is D, a secure string, because it is not a method used to keep secrets out of Terraform configuration files. Options A, B, and C are viable methods.

Let's break down why:

**A. A Terraform provider:** Providers are plugins that enable Terraform to interact with APIs, and they often handle authentication and authorization. While the provider itself might store credentials in its own configuration, the configuration files avoid direct hardcoding of secrets. Instead, provider configurations might reference environment variables or use mechanisms like shared credentials files.

**B. Environment variables:** Environment variables are a well-established and secure way to pass sensitive information to applications, including Terraform. Instead of embedding API keys or passwords in the code, Terraform can retrieve them from environment variables set outside the configuration. This separation of secrets from source code reduces the risk of accidental exposure.

**C. A -var flag:** The -var flag, used in conjunction with the terraform apply or terraform plan commands, allows passing variable values during execution. This is a common technique for injecting secrets dynamically. Rather than directly embedding secrets in variable definition files, their values can be provided as command-line arguments, which are generally not stored in version control systems.

**D. Secure string:** A "secure string" itself isn't a mechanism Terraform natively uses to handle secrets within configuration files. While the concept of securely storing strings is vital, this term more generally applies to mechanisms employed by underlying operating systems or specific applications to handle sensitive string data. It is not a way to abstract secrets from Terraform configuration files. It is a generic method that Terraform might utilize behind the scenes, but not a method for referencing secrets from within configuration.In summary, options A, B, and C offer ways to keep secret values out of Terraform configuration files, while secure string is a general security concept, not a method that accomplishes the stated goal within Terraform.

**Authoritative Links:**

**Terraform Variables:** https://developer.hashicorp.com/terraform/language/values/variables
**Terraform Environment Variables:** https://developer.hashicorp.com/terraform/cli/config/environment-variables
**Terraform Providers:** https://developer.hashicorp.com/terraform/language/providers
**Best Practices for Secrets Management:** https://developer.hashicorp.com/terraform/tutorials/security/secrets

**Question: 44** CertyIQ

What is one disadvantage of using dynamic blocks in Terraform?

A. They cannot be used to loop through a list of values
B. Dynamic blocks can construct repeatable nested blocks
C. They make configuration harder to read and understand
D. Terraform will run more slowly

**Answer: C**

**Explanation:**

The correct answer is C, that dynamic blocks can make configuration harder to read and understand. Dynamic blocks in Terraform are powerful for generating repeatable, nested configuration blocks based on lists or maps. However, their inherent complexity can reduce code readability. Nesting dynamic blocks within each other or using complex conditional logic within them can obscure the intended configuration structure, making it harder for users to quickly grasp the resource definitions and relationships. This complexity introduces cognitive overhead and can increase the likelihood of configuration errors and make debugging more difficult. Although dynamic blocks offer flexibility, overuse or overly complicated usage can counteract Terraform's goal of declarative, easily understandable infrastructure as code. Options A, B, and D are incorrect because dynamic blocks are specifically designed to iterate over lists and maps to create nested blocks. Furthermore, while complex configurations could potentially impact speed, it is not the primary disadvantage inherent to dynamic blocks themselves.Authoritative Links:

Terraform Dynamic Blocks Documentation
Best Practices for Terraform (Specifically regarding readability)

**Question: 45** CertyIQ

Only the user that generated a plan may apply it.

A. True
B. False

**Answer: B**

**Explanation:**

The statement "Only the user that generated a plan may apply it" is **false** in the context of HashiCorp's tools like Terraform. While it's a good practice to maintain clear ownership and traceability, Terraform does not enforce this limitation. Terraform plans are essentially data representations of proposed infrastructure changes, stored in a .tfplan file. This file can be shared and applied by any user with the necessary access and permissions to the underlying infrastructure and Terraform state.

Terraform uses the state file to track the existing infrastructure. As long as a user has access to the state file and the necessary provider credentials, they can apply a plan that was generated by another user. Collaboration within teams often relies on this capability. Teams may assign infrastructure change creation to one team member, and a designated member, perhaps with higher privileges, might perform the application. The ability to separate planning and applying also allows for review processes and checks before infrastructure changes are made, improving governance.

Therefore, the ability to apply a plan isn't restricted to the user who generated it, making the original statement incorrect. This flexibility enhances collaboration and supports various operational models. For

further details, you can consult the official Terraform documentation on plans and state management:

**Terraform Plans:** https://www.terraform.io/docs/cli/commands/plan.html
**Terraform State:** https://www.terraform.io/docs/language/state/

**Question: 46** CertyIQ

Examine the following Terraform configuration, which uses the data source for an AWS AMI.
What value should you enter for the ami argument in the AWS instance resource?

```
data "aws_ami" "ubuntu" {
   ...
}

resource "aws_instance" "web" {
   ami = _____________________
   instance_type = "t2.micro"

   tags = {
      Name = "HelloWorld"
   }
}
```

A. aws_ami.ubuntu
B. data.aws_ami.ubuntu
C. data.aws_ami.ubuntu.id
D. aws_ami.ubuntu.id

**Answer: C**

**Explanation:**
resource "aws_instance" "web"
ami = data.aws_ami.ubuntu.id

Reference:
https://registry.terraform.io/providers/hashicorp/aws/latest/docs/resources/instance

**Question: 47** CertyIQ

FILL BLANK -
You need to specify a dependency manually.
What resource meta-parameter can you use to make sure Terraform respects the dependency?
Type your answer in the field provided. The text field is not case-sensitive and all variations of the correct answer are accepted.

**Answer: depends_on**

## Question: 48

CertyIQ

You have never used Terraform before and would like to test it out using a shared team account for a cloud provider. The shared team account already contains
15 virtual machines (VM). You develop a Terraform configuration containing one VM, perform terraform apply, and see that your VM was created successfully.
What should you do to delete the newly-created VM with Terraform?

A. The Terraform state file contains all 16 VMs in the team account. Execute terraform destroy and select the newly-created VM.

B. The Terraform state file only contains the one new VM. Execute terraform destroy.

C. Delete the Terraform state file and execute Terraform apply.

D. Delete the VM using the cloud provider console and terraform apply to apply the changes to the Terraform state file.

**Answer: B**

**Explanation:**

The correct answer is **B: The Terraform state file only contains the one new VM. Execute terraform destroy.** Here's why:

Terraform manages infrastructure through a state file. This file acts as a mapping between the configuration code and the real-world infrastructure resources. Crucially, Terraform only tracks resources that it created itself. In this scenario, the team's 15 existing VMs were created outside of Terraform's management. Therefore, the initial terraform apply only added the newly created VM to the state file.

Option A is incorrect because the state file wouldn't contain the 15 pre-existing VMs. Option C is wrong because deleting the state file would erase Terraform's knowledge of any resource, including the one you want to delete. Executing terraform apply after that would attempt to recreate the VM. Option D is also incorrect because manually deleting resources outside of Terraform makes the state file out of sync. Then, running terraform apply would simply recreate the resource that was manually deleted since the state file would still expect it to exist.

Executing terraform destroy will remove the infrastructure listed in the state file, which, in this case, is only the one newly created VM. Terraform will understand from the state file that this VM was created by it and is now ready for deletion. terraform destroy is the intended way to remove Terraform-managed resources that are no longer required.

**Authoritative Links for Further Research:**

**Terraform State Documentation:** https://www.terraform.io/language/state - This page provides comprehensive information about how Terraform state works and how it is used for resource management.
**Terraform Destroy Documentation:** https://www.terraform.io/cli/commands/destroy - This page details the functionality of the terraform destroy command.
**Terraform Core Concepts:** https://developer.hashicorp.com/terraform/tutorials/aws-get-started/terraform-concepts - This tutorial explains the core concepts of Terraform, including the state file and its importance.

## Question: 49

CertyIQ

What is the name assigned by Terraform to reference this resource?

```
resource "azurerm_resource_group" "dev" {
  name = "test"
  location = "westus"
}
```

A. dev
B. azurerm_resource_group
C. azurerm
D. test

**Answer: A**

**Explanation:**

Correct answer is A:dev

**Question: 50** CertyIQ

Setting the TF_LOG environment variable to DEBUG causes debug messages to be logged into syslog.

A. True
B. False

**Answer: B**

**Explanation:**

The statement that setting the TF_LOG environment variable to DEBUG causes debug messages to be logged into syslog is incorrect. The TF_LOG variable in Terraform primarily controls the verbosity of logging output to the standard error stream (stderr), not syslog. Setting TF_LOG to DEBUG enables detailed debugging information to be printed on the terminal where Terraform commands are executed, offering granular insights into Terraform's operations, including resource interactions with cloud providers. This detailed logging is beneficial for diagnosing issues within Terraform configurations and understanding the plan and apply processes. Terraform's logging system operates independently of system-level logging mechanisms like syslog. Syslog typically captures system events, not application-specific debug output. To forward logs to syslog or other log aggregation systems, additional configurations specific to the environment and the chosen logging platform must be implemented. This is generally achieved using dedicated logging agents or custom scripts that capture Terraform's standard output and forward it appropriately. While Terraform does have a mechanism to export logs using TF_LOG_PATH, this simply directs the log output to a file, and does not interact with syslog. Therefore, the core function of TF_LOG is to directly display detailed output on stderr or save it in file depending on additional settings, but does not automatically integrate with syslog, thus making the original assertion false. For further exploration on Terraform's logging, refer to the official documentation here: https://developer.hashicorp.com/terraform/cli/config/environment-variables#tf_log.

**Question: 51** CertyIQ

Where in your Terraform configuration do you specify a state backend?

A. The terraform block
B. The resource block
C. The provider block
D. The datasource block

**Answer: A**

**Explanation:**

The correct answer is **A. The terraform block.** The Terraform configuration uses the terraform block to define settings that pertain to the overall project, not specific resources or providers. This is where the state backend configuration resides. A state backend is essential for managing Terraform's state, which tracks the resources provisioned by the configuration. The state file stores critical information needed for Terraform to understand which resources exist in the target infrastructure, as well as any dependencies between resources. When working collaboratively or across multiple development environments, a remote backend, like AWS S3 or Azure Storage, is necessary. The terraform block is also used to specify the required Terraform version and providers. Resource blocks define specific infrastructure components. Provider blocks specify the interface to an infrastructure provider such as AWS or Azure. Datasource blocks fetch information on existing resources. Therefore, only the terraform block is the logical location to specify the state backend configuration. Configuring the backend ensures consistent state management, collaboration, and safeguards against accidental state overwrite or loss. The backend configuration inside terraform block specifies where Terraform should store the state.

Further research:

Terraform Configuration: Terraform Block
Terraform Backends

**Question: 52** CertyIQ

In Terraform 0.13 and above, outside of the required_providers block, Terraform configurations always refer to providers by their local names.

A. True
B. False

**Answer: A**

**Explanation:**

The statement is indeed **True**. In Terraform versions 0.13 and later, providers are referenced by their local names within the configuration, excluding the required_providers block. This block, found in the terraform configuration, specifies the source and version of providers but doesn't assign the names used elsewhere. Instead, within resource blocks and other provider-dependent components, you use an alias or the provider argument to refer to a specific provider instance using a name defined by the user.

Prior to Terraform 0.13, providers were often implicitly referred to based on their name on the Terraform Registry, leading to potential ambiguity when working with multiple instances of the same provider. The move to local names promotes clarity and avoids naming conflicts, particularly when configuring multiple accounts or regions for the same provider. For instance, you might define two AWS provider configurations, one for a 'prod' environment and one for a 'dev' environment, each assigned a unique local name. The local name acts as a namespace, differentiating the provider instances used within the Terraform configuration. This allows each resource block to explicitly state which provider instance it should use, enhancing modularity and reducing potential errors. By decoupling the provider's registry name from its local identifier, Terraform achieves a

more flexible and predictable configuration management approach.

Refer to the following documentation for more in-depth information:

**Terraform Providers Documentation:** https://www.terraform.io/docs/language/providers/index.html
**Terraform Required Providers:** https://www.terraform.io/docs/language/providers/requirements.html
**Terraform Provider Configuration:** https://www.terraform.io/docs/language/providers/configuration.html

**Question: 53** CertyIQ

What command should you run to display all workspaces for the current configuration?

A. terraform workspace
B. terraform workspace show
C. terraform workspace list
D. terraform show workspace

**Answer: C**

**Explanation:**

The correct command to list all Terraform workspaces associated with the current configuration is terraform workspace list. Workspaces in Terraform provide a mechanism to manage multiple, isolated environments or feature sets within a single configuration. This allows teams to maintain separate states for development, staging, and production without needing to duplicate code. terraform workspace by itself provides a list of options, but not the list of workspaces themselves. terraform workspace show displays the name of the currently selected workspace, not all available workspaces. terraform show workspace is an invalid command. terraform workspace list, specifically, is designed to display all existing workspaces associated with the configuration, making it the direct and accurate solution to the problem. This ensures the user can see all available environments and understand the workspace landscape.

For further research, refer to the official Terraform documentation on workspaces:https://www.terraform.io/cli/commands/workspacehttps://developer.hashicorp.com/terraform/cli/comm

**Question: 54** CertyIQ

Terraform providers are always installed from the Internet.

A. True
B. False

**Answer: B**

**Explanation:**

The statement "Terraform providers are always installed from the Internet" is false. While the default behavior of Terraform is to download providers from the HashiCorp Registry over the internet, this is not the only way providers can be acquired. Terraform offers mechanisms for using locally stored providers, allowing for environments where internet access is restricted or where specific provider versions are preferred. These local providers can be placed within specific directories that Terraform searches or installed using a local registry mirroring mechanism. This is crucial for air-gapped environments or organizations adhering to strict security policies. Furthermore, users might want to build custom providers and load them directly, bypassing

the internet entirely. Therefore, relying solely on internet-based downloads isn't mandatory. This flexibility enables Terraform to be adaptable to various network configurations and compliance needs. For further reading on how to specify local providers and utilize a local registry mirror, please refer to the official Terraform documentation: https://www.terraform.io/language/providers/configuration and https://www.terraform.io/cli/config/config-file

**Question: 55** CertyIQ

Which of these is the best practice to protect sensitive values in state files?

A. Blockchain
B. Secure Sockets Layer (SSL)
C. Enhanced remote backends
D. Signed Terraform providers

**Answer: C**

**Explanation:**

The correct answer is C, Enhanced remote backends, because it directly addresses the need to secure sensitive data within Terraform state files. While Terraform does not inherently encrypt state files by default, remote backends offer mechanisms to do so. These mechanisms involve storing the state in a secure, often encrypted, environment, like cloud storage (e.g., AWS S3, Azure Storage, Google Cloud Storage) with encryption at rest and transit. This ensures that even if the underlying storage is compromised, the sensitive information within the state file remains protected. Options A, Blockchain, is not relevant to Terraform state management and its security. Blockchain is a decentralized ledger technology, not designed for state file protection. Option B, Secure Sockets Layer (SSL), now commonly known as Transport Layer Security (TLS), secures communication channels but doesn't protect state file contents at rest. While important for secure network traffic, SSL/TLS does not directly address state file encryption. Option D, Signed Terraform providers, improves the integrity of provider binaries but doesn't safeguard values stored in state files. Therefore, the use of enhanced remote backends that support encryption is the most appropriate approach to protect sensitive values within Terraform state files. For further details, refer to HashiCorp's documentation on remote state and state locking: https://www.terraform.io/docs/language/state/remote.html and on securing state: https://www.terraform.io/language/state/sensitive-data

**Question: 56** CertyIQ

When does terraform apply reflect changes in the cloud environment?

A. Immediately
B. However long it takes the resource provider to fulfill the request
C. After updating the state file
D. Based on the value provided to the -refresh command line argument
E. None of the above

**Answer: B**

**Explanation:**

The correct answer is **B. However long it takes the resource provider to fulfill the request.** Terraform's apply

command does not directly manipulate cloud resources. Instead, it sends API calls to the respective cloud provider's API based on the desired state defined in the Terraform configuration. The cloud provider then processes these requests to create, modify, or delete resources. The time it takes for these changes to propagate and be reflected in the cloud environment is entirely dependent on the provider's implementation and the complexity of the resources being managed. Terraform's state file is updated only after the cloud provider confirms the successful application of changes. Therefore, the state file update is a consequence of successful cloud resource provisioning, not the cause of the changes. Option A is incorrect because changes are not immediate. Option C is incorrect because the state file update is not what causes the change. Option D is incorrect as the -refresh flag updates the state file but does not affect how quickly the cloud provider applies the changes. The actual application time can vary significantly depending on the cloud provider, the type of resource, and its configuration. This asynchronous process highlights the role of resource providers as the critical element in delivering real-world cloud changes based on Terraform instructions.

**Supporting Links:**

**Terraform Apply Command:** https://developer.hashicorp.com/terraform/cli/commands/apply - This official documentation explains the apply command's function and lifecycle.
**Terraform State:** https://developer.hashicorp.com/terraform/language/state - This official documentation details the state file's role and its relationship with cloud resources.
**Provider Documentation (Example: AWS Provider):**
https://registry.terraform.io/providers/hashicorp/aws/latest/docs - Specific providers provide details on their API interactions and how long creation or modification requests generally take. The overall flow is that terraform makes a request and the provider handles the communication with the cloud resource provider api, which takes the necessary time to execute.

**Question: 57** CertyIQ

How would you reference the "name" value of the second instance of this fictitious resource?

```
resource "aws_instance" "web" {
  count = 2
  name = "terraform-${count.index}"
}
```

A. element(aws_instance.web, 2)
B. aws_instance.web[1].name
C. aws_instance.web[1]
D. aws_instance.web[2].name
E. aws_instance.web.*.name

**Answer: B**

**Explanation:**

B - aws_instance.web[1].name !!!

Index starts at 0 so the second instance would be 1 - the link below confirms this:

https://www.terraform.io/language/meta-arguments/count#referring-to-instances

**Question: 58** CertyIQ

A Terraform provider is not responsible for:

A. Understanding API interactions with some service
B. Provisioning infrastructure in multiple clouds
C. Exposing resources and data sources based on an API
D. Managing actions to take based on resource differences

**Answer: B**

**Explanation:**

The correct answer is **B. Provisioning infrastructure in multiple clouds.** Here's why:

Terraform providers act as a bridge between Terraform and various infrastructure platforms or services. They are responsible for understanding the APIs of those services (A), exposing their resources and data sources (C), and managing the actions needed to reconcile the desired state with the actual state of those resources (D). In essence, a provider allows Terraform to interact with and manage the resources of a specific service.

While Terraform can manage resources across multiple clouds using different providers (e.g., aws, google, azure providers), it's the responsibility of each individual provider to handle only the specifics of its designated service, not to provision across multiple diverse platforms. A single provider is typically tailored to a single API of a specific service, cloud, or platform. This architecture allows for modularity and reduces complexity. Terraform's power lies in its ability to orchestrate different providers and resources within a single configuration, but not within the same provider itself.

Therefore, a Terraform provider is not designed or intended to be able to provision resources in multiple cloud environments directly; it's designed to manage one specific service. It's the responsibility of the Terraform user, through configuration, to utilize different providers to orchestrate resources across different environments.

**Authoritative Links:**

**HashiCorp Terraform Providers:** https://www.terraform.io/docs/providers/index.html
**Terraform Provider Development:** https://www.terraform.io/docs/extend/index.html

**Question: 59** CertyIQ

Terraform provisioners can be added to any resource block.

A. True
B. False

**Answer: A**

**Explanation:**

The statement is **True**. Terraform provisioners are indeed designed to be included within any resource block. Their primary function is to execute actions on the resource after its creation (or modification) by Terraform. This post-creation processing can encompass a wide range of activities, from running scripts to deploying software, or configuring the resource further. The flexibility of provisioners lies in their ability to interact with the infrastructure being managed, extending Terraform's core declarative capabilities with imperative steps. While provisioners can add complexity and should be used judiciously, their placement within resource blocks enables targeted actions directly associated with specific resources. This direct association ensures that

post-creation steps are correctly applied only to the intended resources. Common use cases include installing packages on a virtual machine or running database initialization scripts. However, it's important to note that provisioners are executed locally, on the machine where Terraform is run, unless a remote-exec provisioner is used.

For more in-depth information, refer to the official Terraform documentation:

https://developer.hashicorp.com/terraform/language/resources/provisioners/syntaxhttps://developer.hashicorp.co

**Question: 60** CertyIQ

What is terraform refresh intended to detect?

A. Terraform configuration code changes
B. Empty state files
C. State file drift
D. Corrupt state files

**Answer: C**

**Explanation:**

The correct answer is C, **State file drift**. terraform refresh is specifically designed to reconcile the Terraform state file with the current state of the infrastructure in the real world. It doesn't analyze configuration code changes (A), which is the job of terraform plan and terraform apply. It's also not concerned with empty state files (B), which are typically an initialization issue, nor corrupt state files (D), which require separate handling.

State file drift refers to discrepancies between the declared infrastructure in the Terraform state file and the actual infrastructure deployed. This can happen when changes are made outside of Terraform, like manual modifications via the cloud provider's console or CLI, or by other tools. terraform refresh queries the cloud provider APIs to determine the current status of the resources tracked by Terraform and updates the state file accordingly. It identifies changes like modifications to resource properties, deletion, or the creation of external dependencies. By detecting these out-of-band changes, terraform refresh ensures that the Terraform state reflects the real-world environment, preparing for subsequent plans and applications. This process doesn't make any changes to the infrastructure itself; it only updates the state.

Therefore, terraform refresh is crucial for maintaining synchronization between your desired infrastructure state and the actual state, which is essential for consistent and predictable infrastructure management with Terraform.

Relevant resources:

Terraform refresh documentation
Terraform state management

**Question: 61** CertyIQ

FILL BLANK -
Which flag would you add to terraform plan to save the execution plan to a file?
Type your answer in the field provided. The text field is not case-sensitive and all variations of the correct answer are accepted.

**Answer: -out=FILENAME**

**Explanation:**

Reference:
https://www.terraform.io/docs/cli/commands/plan.html
" target="_blank" style="word-break: break-all;">

You can use the optional `-out=FILE` option to save the generated plan to a file on disk, which you can later execute by passing the file to `terraform apply` as an extra argument. This two-step workflow is primarily intended for when running Terraform in automation.

If you run `terraform plan` without the `-out=FILE` option then it will create a *speculative plan*, which is a description of the effect of the plan but without any intent to actually apply it.

**Question: 62** CertyIQ

FILL BLANK -
What is the name of the default file where Terraform stores the state?
Type your answer in the field provided. The text field is not case-sensitive and all variations of the correct answer are accepted.

**Answer:**

Terraform.tfstate

**Explanation:**

Reference:
https://www.terraform.io/docs/language/state/index.html
" target="_blank" style="word-break: break-all;">

# State

JUMP TO SECTION

Terraform must store state about your managed infrastructure and configuration. This state is used by Terraform to map real world resources to your configuration, keep track of metadata, and to improve performance for large infrastructures.

This state is stored by default in a local file named "terraform.tfstate", but it can also be stored remotely, which works better in a team environment.

**Question: 63** CertyIQ

A Terraform local value can reference other Terraform local values.

A. True
B. False

**Answer: A**

**Explanation:**

The statement that Terraform local values can reference other local values is indeed true. Terraform's local values mechanism allows you to define reusable expressions within your configuration. These local values can be based on simple string concatenations, calculations, or, crucially, references to other local values. This enables the creation of complex data structures derived from simpler, foundational elements, enhancing the organization and maintainability of your Terraform code. For example, you could have local.region and then derive local.availability_zones based on the region. This cascading effect of dependencies promotes modularity and reduces redundancy. If the value of local.region were to change, dependent local values such as local.availability_zones would also update. This feature encourages DRY (Don't Repeat Yourself) principles in Infrastructure as Code. While a cycle (circular dependency) isn't allowed, a linear or tree-like dependency structure among locals is fully supported. This capability allows for the construction of more complex configurations based on intermediate variables, without needing to introduce unnecessary resources or hardcoded values. Local variables are evaluated within the context of the same module where they're declared. Therefore, this referencing is confined to the current module scope. This feature also improves readability since complex values are broken down into meaningful names via these local values.

For further exploration, consult the official Terraform documentation on local values:
https://www.terraform.io/language/values/locals

**Question: 64** CertyIQ

Which of the following is not a valid Terraform collection type?

A. list
B. map
C. tree
D. set

**Answer: C**

**Explanation:**

The correct answer is C, "tree," because Terraform natively supports three collection types: lists, maps, and sets. A list in Terraform is an ordered sequence of values, allowing for duplicate entries. Maps, also known as dictionaries in other languages, store data as key-value pairs. Sets, unlike lists, are unordered collections of unique values; duplicates are automatically eliminated. A "tree" is a hierarchical data structure used extensively in computer science, but Terraform doesn't provide it as a built-in collection type. Terraform's data structures are designed to facilitate infrastructure-as-code management, primarily working with single-dimensional or key-value associations, not complex, multi-dimensional tree structures directly. Representing hierarchical data in Terraform often involves a combination of lists, maps, and nested modules instead of a dedicated tree type. Terraform focuses on provisioning cloud resources, not managing graph-like relationships between them within the language itself. You would likely use external tooling or more complex data manipulation before passing it to Terraform for infrastructure configuration. These core collection types are sufficient for organizing most configuration options and data within a Terraform workflow.

For further research, refer to the official Terraform documentation on data types:

Terraform Data Types
Terraform Lists
Terraform Maps

Terraform Sets

**Question: 65** CertyIQ

When running the command terraform taint against a managed resource you want to force recreation upon, Terraform will immediately destroy and recreate the resource.

A. True
B. False

**Answer: B**

**Explanation:**

The statement is false. The terraform taint command does not immediately trigger the destruction and recreation of a resource. Instead, it marks the resource as "tainted" within Terraform's state file. This taint signifies that the resource is considered out of sync with the desired configuration. During the subsequent terraform apply operation, Terraform will recognize the tainted resource and proceed with its destruction and recreation to align with the desired state. This two-step process ensures that changes are not abruptly executed, giving users an opportunity to review the plan before the actual modifications occur. It provides a safety mechanism to prevent accidental disruptions. The command only flags the resource as needing to be replaced. The change will not be put into effect until the next plan/apply. Think of it as setting a flag for later, rather than taking immediate action. Therefore, while the end result will be the resource being recreated, the taint command itself is not the action that accomplishes this, making option B, False, the correct answer. This approach is in line with Terraform's principles of declarative configuration management, planning before executing. This avoids any surprises, and provides transparency.

**Authoritative Links:**

**Terraform documentation on taint:** https://www.terraform.io/cli/commands/taint
**Terraform resource lifecycle:** https://developer.hashicorp.com/terraform/language/resources/lifecycle

**Question: 66** CertyIQ

All standard backend types support state storage, locking, and remote operations like plan, apply and destroy.

A. True
B. False

**Answer: B**

**Explanation:**

The statement that all standard Terraform backend types support state storage, locking, and remote operations (plan, apply, destroy) is false. While most commonly used backends like s3, azurerm, gcs, and terraform cloud provide these features, not all do. Specifically, the local backend, a fundamental example, stores state on the local filesystem and doesn't inherently support state locking or remote operations. Locking is essential for collaboration to prevent concurrent modifications that could corrupt state. Remote operations facilitate team workflows and access control, which are absent in the local backend's design. The choice of backend significantly impacts collaboration possibilities and operational reliability. Backends like http or artifactory often have specific usage scenarios and may require custom configurations to enable features like state locking. Furthermore, some custom backends built with the provider SDK might lack all three stated features. This distinction highlights that Terraform's backend is modular, with varying

functionality across types. Relying on the assumption that every backend type offers all the described features can lead to operational issues, emphasizing the importance of understanding the capabilities of each selected backend. Thus, not all standard backend types inherently support state storage, locking, and remote operations, particularly those designed for simpler, non-collaborative or testing scenarios.

Authoritative Links:

**Terraform Backends:** https://www.terraform.io/docs/language/settings/backends/index.html
**Terraform Local Backend:** https://www.terraform.io/docs/language/settings/backends/local.html
**Terraform State Locking:** https://www.terraform.io/docs/language/state/locking.html

**Question: 67** CertyIQ

How can terraform plan aid in the development process?

A. Validates your expectations against the execution plan without permanently modifying state
B. Initializes your working directory containing your Terraform configuration files
C. Formats your Terraform configuration files
D. Reconciles Terraform's state against deployed resources and permanently modifies state using the current status of deployed resources

**Answer: A**

**Explanation:**

The correct answer is A because terraform plan's primary function is to generate an execution plan, detailing the proposed changes to infrastructure based on the current configuration and state. Critically, this plan is displayed for review before any actual modifications occur. This allows developers to validate their intended infrastructure changes without immediately applying them, offering a safety net against unintended consequences. It allows teams to inspect additions, removals, or modifications of resources, fostering a better understanding of the impact of code changes. This process enables early identification of errors or misconfigurations, significantly reducing risks associated with infrastructure deployment. Unlike options B and C, which involve initialization and formatting respectively, terraform plan centers around anticipating changes, not managing the workspace or code style. Option D's description relates more closely to terraform apply, which does reconcile state and modify resources, unlike the preview-focused terraform plan. Therefore, terraform plan is a crucial tool for safe and controlled infrastructure development, allowing for review and confirmation before any actual state modification occurs.

**Authoritative Links:**

**Terraform Documentation on terraform plan:** https://www.terraform.io/cli/commands/plan - This is the official documentation explaining the command and its use.
**HashiCorp Learn Platform on Planning:** https://learn.hashicorp.com/tutorials/terraform/plan - A guided tutorial showing how planning works within a Terraform workflow.

**Question: 68** CertyIQ

You would like to reuse the same Terraform configuration for your development and production environments with a different state file for each.
Which command would you use?

A. terraform import
B. terraform workspace

C. terraform state

D. terraform init

**Answer: B**

**Explanation:**

The correct answer is **B. terraform workspace**.

Terraform workspaces are designed to manage multiple, distinct environments (like development and production) using the same configuration. Each workspace maintains its own isolated state file, allowing you to deploy and manage infrastructure changes independently in each environment without conflicts. The terraform workspace command provides functionalities to create, list, select, and delete workspaces, thus enabling environment segregation. Using different state files is crucial for preventing unintended modifications or corruptions of production resources when working on development environments. Options A, C, and D, while Terraform commands, do not directly address the requirement of multiple state files from the same configuration for different environments. terraform import is used for bringing existing infrastructure under Terraform management. terraform state manages the state of already created resources. terraform init is for initializing a Terraform working directory. For further understanding, please refer to the official Terraform documentation on workspaces: https://www.terraform.io/docs/language/state/workspaces.html. This feature aligns with infrastructure-as-code best practices, promoting maintainability and reducing the risk of errors in diverse environments.

**Question: 69** CertyIQ

What is the name assigned by Terraform to reference this resource?

```
mainresource “google_compute_instance” “main” {
        name = “test”
}
```

A. compute_instance

B. main

C. google

D. teat

**Answer: B**

**Explanation:**

Main

is the name of the resources

**Question: 70** CertyIQ

You're building a CI/CD (continuous integration/ continuous delivery) pipeline and need to inject sensitive variables into your Terraform run.
How can you do this safely?

A. Pass variables to Terraform with a "var flag

B. Copy the sensitive variables into your Terraform code

C. Store the sensitive variables in a secure_vars.tf file

D. Store the sensitive variables as plain text in a source code repository

**Answer: A**

**Explanation:**

The most secure approach to injecting sensitive variables into a Terraform run during CI/CD is by using the -var flag (Option A). This allows you to pass variables as command-line arguments, often sourced from secure environment variables within your CI/CD system. This method avoids hardcoding sensitive values directly into your Terraform configuration, which is highly insecure (Options B and C). Storing variables in a secure_vars.tf file (Option C) still exposes these values to version control and anyone with access. Similarly, keeping sensitive information as plain text in a repository (Option D) is a significant security risk. Environment variables are typically managed by the CI/CD system with proper access controls, ensuring only authorized pipelines and users can access these values. The -var flag then injects these values during Terraform plan and apply stages without them being persisted within the configuration itself. Using the -var flag with environment variables is a standard and recommended security practice, aligned with principles like the principle of least privilege and separation of concerns, where configuration is decoupled from sensitive information. This method minimizes exposure by keeping secrets outside of source code, reducing potential breaches. For further research, consult the Terraform documentation regarding input variables: https://www.terraform.io/docs/language/values/variables.html and the guidelines on managing sensitive data with Terraform: https://developer.hashicorp.com/terraform/tutorials/configuration/sensitive-data.

**Question: 71** CertyIQ

Your security team scanned some Terraform workspaces and found secrets stored in a plaintext in state files. How can you protect sensitive data stored in Terraform state files?

A. Delete the state file every time you run Terraform

B. Store the state in an encrypted backend

C. Edit your state file to scrub out the sensitive data

D. Always store your secrets in a secrets.tfvars file.

**Answer: B**

**Explanation:**

The correct answer is **B. Store the state in an encrypted backend.** This is the best practice for securing sensitive data within Terraform state files. Terraform state files inherently contain configuration information, including potentially sensitive data like passwords, API keys, and database connection strings. Plaintext storage of state files poses a significant security risk, as unauthorized access could lead to full system compromise.

Option A, deleting the state file, would lead to data loss and render Terraform unable to manage the infrastructure, thus being not only impractical, but disruptive. Option C, manually editing the state file, is error-prone, insecure, and would likely corrupt the state, causing severe management issues. Option D, storing secrets in secrets.tfvars, is a helpful practice for injecting secrets, but it does not solve the problem of secrets being written into the state file in plain text if not configured properly.

Terraform backends provide a mechanism for storing state files remotely and can be configured to use encryption at rest and in transit. Encrypted backends like AWS S3 with server-side encryption, Azure Storage with encryption at rest, or HashiCorp Consul/Terraform Cloud offer a strong layer of protection, ensuring that

even if the backend storage is breached, the state information remains encrypted and unreadable without the appropriate encryption keys. This mitigates the risk of information exposure.

Therefore, choosing an encrypted backend is crucial for securing Terraform state files and protecting sensitive information.

For further research, refer to these authoritative sources:

**Terraform Backends Documentation:** https://www.terraform.io/docs/language/settings/backends/index.html
**Terraform Cloud Documentation:** https://www.terraform.io/cloud-docs/index
**AWS S3 Server-Side Encryption:** https://docs.aws.amazon.com/AmazonS3/latest/userguide/serv-side-encryption.html
**Azure Storage Encryption:** https://learn.microsoft.com/en-us/azure/storage/common/storage-service-encryption

**Question: 72** CertyIQ

In contrast to Terraform Open Source, when working with Terraform Enterprise and Cloud Workspaces, conceptually you could think about them as completely separate working directories.

A. True
B. False

**Answer: A**

**Explanation:**

The statement is indeed true. Terraform Enterprise and Cloud Workspaces introduce a paradigm shift compared to using Terraform Open Source locally. In the open-source model, a single working directory typically houses your Terraform configuration, state files, and other associated resources. However, with Workspaces in Terraform Enterprise and Cloud, each workspace acts as an isolated environment. This isolation means that each workspace has its own dedicated working directory, independent of others. This compartmentalization is crucial for managing multiple infrastructure deployments or managing infrastructure in different environments (e.g., development, staging, production) separately. Changes in one workspace do not directly affect another, promoting stability and preventing accidental cross-contamination. Each workspace stores its own Terraform state, configurations, and potentially its own variables. This concept helps achieve team-based collaboration and simplifies infrastructure management in larger and more complex projects. It allows different teams or individuals to work on different aspects of the infrastructure concurrently without interference. Think of it as individual, secured "sandboxes" for infrastructure deployment. This architecture adheres to the cloud-native principle of isolation and enables greater control over deployments. This contrasts sharply with the single directory model of Terraform Open Source, where the lack of isolation can lead to conflicts and unintentional modifications. The use of dedicated workspaces promotes organization and maintainability in an enterprise setting.

Relevant resources:

**Terraform Cloud Workspaces Documentation:** https://developer.hashicorp.com/terraform/cloud-docs/workspaces
**Terraform Enterprise Documentation:** https://developer.hashicorp.com/terraform/enterprise

**Question: 73** CertyIQ

You want to know from which paths Terraform is loading providers referenced in your Terraform configuration (*.tf

files). You need to enable debug messages to find this out.
Which of the following would achieve this?

A. Set the environment variable TF_LOG=TRACE
B. Set verbose logging for each provider in your Terraform configuration
C. Set the environment variable TF_VAR_log=TRACE
D. Set the environment variable TF_LOG_PATH

**Answer: A**

**Explanation:**

The correct answer is A: Set the environment variable TF_LOG=TRACE. This is because Terraform's built-in logging mechanism utilizes the TF_LOG environment variable to control the verbosity of its output. Setting TF_LOG to TRACE enables the most detailed level of logging, which includes information about provider loading paths, as well as a myriad of other debugging details. This level of detail is crucial for diagnosing issues related to provider discovery and usage. Options B, C, and D are incorrect. B is not feasible as Terraform does not offer a per-provider verbose logging configuration setting. C is incorrect as TF_VAR_log is not a valid Terraform environment variable. D is misleading as TF_LOG_PATH configures where the log output is written but not the verbosity level of the log. Therefore, TF_LOG=TRACE is the specific and correct approach for achieving the desired level of debugging output regarding provider loading paths. This directly addresses the need for detailed output to resolve the problem.

For further research on Terraform's logging environment variables, consult the official Terraform documentation at https://www.terraform.io/cli/config/environment-variables.

**Question: 74** CertyIQ

How is terraform import run?

A. As a part of terraform init
B. As a part of terraform plan
C. As a part of terraform refresh
D. By an explicit call
E. All of the above

**Answer: D**

**Explanation:**

The correct answer is **D. By an explicit call**. Terraform import is not an automatic process that occurs during terraform init, terraform plan, or terraform refresh. It is a distinct operation that requires a specific terraform import command to be executed by the user. This command explicitly instructs Terraform to take an existing resource, that was not initially created by Terraform, and bring it under Terraform's management. terraform init primarily initializes the working directory and downloads provider plugins. terraform plan analyzes the current configuration against the existing infrastructure to determine what changes need to be applied. terraform refresh updates the state file with the current state of the infrastructure but doesn't change the configuration or import resources. The terraform import command requires specific resource addresses (defined in configuration files) and IDs from the provider to map the existing resources to the Terraform code. It allows you to begin managing existing infrastructure, effectively bridging the gap between infrastructure created outside of Terraform and Terraform's management capabilities. It is not a routine part of a typical terraform apply workflow or other Terraform stages and hence needs to be explicitly invoked by the user.

For further research, refer to the official Terraform documentation:
https://www.terraform.io/docs/cli/commands/import.html

**Question: 75** CertyIQ

You have a simple Terraform configuration containing one virtual machine (VM) in a cloud provider. You run terraform apply and the VM is created successfully.
What will happen if you delete the VM using the cloud provider console, and run terraform apply again without changing any Terraform code?

A. Terraform will remove the VM from state file
B. Terraform will report an error
C. Terraform will not make any changes
D. Terraform will recreate the VM

**Answer: D**

**Explanation:**

Here's a detailed justification for why option D, "Terraform will recreate the VM," is the correct answer:

Terraform's core function is to manage infrastructure state based on configurations defined in code. When you initially ran terraform apply, Terraform created the VM and recorded its details (resource ID, properties, etc.) in its state file. This file acts as a source of truth about the managed infrastructure.

When you delete the VM through the cloud provider's console, Terraform's state file still believes the VM exists. When you run terraform apply again, Terraform compares the desired state (defined in your code) with the current state (recorded in the state file and then refreshed by fetching real resource data). It discovers that the VM, which should exist according to the state, is missing.

To reconcile this discrepancy, Terraform must take action. Because the desired state includes the VM resource, Terraform's plan will show that it needs to create a new VM to match the declared configuration, then the apply command will execute this. It essentially tries to match the desired state with actual infrastructure by creating a missing resource. This process isn't about detecting manual deletions or actively monitoring outside influences; it's about bringing the real world infrastructure in line with the state file.

Options A, B, and C are incorrect. Terraform will not remove the resource from the state file because it still thinks the resource is present in the real infrastructure until a plan and apply command has updated the state to reflect reality. Option B is incorrect because Terraform will not throw an error in this scenario; it will identify that a change to the state is needed and generate a plan and apply the necessary actions to achieve the required state. Option C is wrong because Terraform will change the actual infrastructure since there is a difference between the state file and the real infrastructure.

Therefore, the correct answer is D: Terraform will recreate the VM. It emphasizes the declarative nature of Terraform, which strives to achieve the desired state irrespective of external modifications.

**Authoritative Links for Further Research:**

**Terraform State Documentation:** https://www.terraform.io/docs/language/state/index.html - Deep dive into how Terraform stores and manages state.
**Terraform Workflow Documentation:** https://www.terraform.io/docs/cli/index.html - Understanding the plan and apply steps
**Terraform Core Concepts:** https://www.terraform.io/docs/intro/index.html - Covers the main functionalities and workflow of Terraform.

**Question: 76** CertyIQ

Which of these options is the most secure place to store secrets foe connecting to a Terraform remote backend?

A. Defined in Environment variables
B. Inside the backend block within the Terraform configuration
C. Defined in a connection configuration outside of Terraform
D. None of above

**Answer: A**

**Explanation:**

The most secure option for storing secrets related to a Terraform remote backend is typically using **environment variables (A)**. While defining secrets directly within the backend block (B) is convenient, it's highly discouraged due to security risks. These secrets would be committed directly into the Terraform configuration files, often version-controlled, making them easily accessible to anyone with access to the repository. A connection configuration outside of Terraform (C) could be an improvement, but it still faces similar issues if not properly secured. Environment variables, on the other hand, offer a better level of separation. They are usually not directly stored alongside the code, reducing the risk of accidental exposure. Furthermore, many CI/CD systems and orchestration tools allow for secure injection of environment variables, further bolstering security. Terraform can retrieve these secrets during runtime, minimizing the chances of them being hardcoded in accessible configuration files. This approach aligns with the principle of least privilege and secure secret management practices. While other more robust secret management tools exist, environment variables, when combined with good security practices, are generally considered more secure than the alternatives presented within the given question context.

**Authoritative links:**

**Terraform Documentation on Backend Configuration:**
https://www.terraform.io/language/settings/backends/configuration (While not explicitly stating environment variables are best for secrets, it highlights how sensitive information is better not hardcoded)
**General Security Best Practices:**

**OWASP Secret Management Cheat Sheet:**
https://cheatsheetseries.owasp.org/cheatsheets/Secret_Management_Cheat_Sheet.html (Highlights the general principle of avoiding hardcoded secrets)
**HashiCorp Vault:** https://www.vaultproject.io/ (While not the answer in this specific context, it highlights best practices for secret management and is an alternative, more secure solution than environment variables in complex setups).

It's important to note that while environment variables are generally a better choice than hardcoding in the backend block, for production environments, using a dedicated secret management solution like HashiCorp Vault or cloud-specific secret management services (AWS Secrets Manager, Azure Key Vault, GCP Secret Manager) is strongly advised for enhanced security and auditability.

**Question: 77** CertyIQ

Your DevOps team is currently using the local backend for your Terraform configuration. You would like to move to a remote backend to begin storing the state file in a central location.
Which of the following backends would not work?

A. Amazon S3
B. Artifactory
C. Git
D. Terraform Cloud

**Answer: C**

**Explanation:**

The correct answer is C, Git, because Git is a version control system, not a state storage backend. Terraform backends are designed to store the state file, which contains a snapshot of your infrastructure's current configuration. This state file is crucial for Terraform to track and manage changes to your infrastructure. Backends like Amazon S3 (A), Terraform Cloud (D), and other similar options like Azure Storage or Google Cloud Storage, are designed to store this state remotely, providing features like locking, versioning, and collaboration. Artifactory (B), while primarily known for artifact management, can also store files and can be configured to act as a Terraform state backend, though it is less common than the other options. Git, on the other hand, stores the code and configuration files and isn't built to manage the state file which is a database in essence; it could lead to race conditions and inconsistencies if used for that purpose. Using Git to store the Terraform state directly is problematic because it lacks necessary features like state locking, which prevents concurrent modifications and can lead to data corruption or inconsistencies. For more information on Terraform backends, consult the official Terraform documentation: https://www.terraform.io/language/settings/backends and for the disadvantages of using a local file, read https://www.terraform.io/language/state/backends#disadvantages-of-local-backend .

**Question: 78** CertyIQ

Which backend does the Terraform CLI use by default?

A. Terraform Cloud
B. Consul
C. Remote
D. Local

**Answer: D**

**Explanation:**

The Terraform CLI, by default, utilizes the "local" backend for storing state files. This backend stores the state directly on the machine where Terraform commands are executed, typically in a file named terraform.tfstate. This is the simplest form of state management, requiring no additional infrastructure. It's suitable for individual developers or small, isolated projects where collaboration is not a priority. However, using the local backend for team-based development is generally not recommended due to potential state file corruption, conflicts during concurrent updates, and lack of version control. Options like Terraform Cloud, Consul, or other remote backends are crucial for teams, offering features like state locking, remote operations, and consistent state storage across collaborators. While these alternatives offer superior collaboration capabilities, the local backend remains the default because of its ease of use for getting started with Terraform. Understanding the default backend is fundamental before progressing to more advanced state management solutions. Refer to the official Terraform documentation for more detail on backends and their configurations https://www.terraform.io/docs/language/state/backends.html. The local backend's convenience is juxtaposed by its limitations in collaborative environments, highlighting why users quickly transition to remote backends in real-world scenarios.

**Question: 79** CertyIQ

When you initialize Terraform, where does it cache modules from the public Terraform Module Registry?

A. On disk in the /tmp directory
B. In memory
C. On disk in the .terraform sub-directory
D. They are not cached

**Answer: C**

**Explanation:**

The correct answer is C: On disk in the .terraform sub-directory. When terraform init is executed, Terraform retrieves necessary providers and modules to manage your infrastructure. Public modules from the Terraform Module Registry are not kept in memory or the temporary /tmp directory. Instead, they're downloaded and cached locally within the .terraform directory, which is automatically created in the current working directory (where your Terraform configuration files reside). This caching mechanism improves subsequent Terraform operations by avoiding redundant downloads. The .terraform directory is specifically designed to hold these downloaded resources, ensuring quicker initialization and execution. Caching modules locally reduces network traffic and speeds up development workflows. This behavior applies regardless of the cloud provider or specific module being used. The cached modules allow Terraform to function offline if the registry becomes temporarily unavailable. These local copies also serve as a source for future module usage as long as the configuration specifies those versions. It's crucial to understand that the .terraform directory should not be committed to version control as it contains environment-specific resources. Maintaining a consistent and predictable infrastructure relies on this localized caching and resource management.

For more information on Terraform module caching, please refer to the following official Terraform documentation:

**Terraform Initialization Process:** https://developer.hashicorp.com/terraform/cli/commands/init
**Module Sources:** https://developer.hashicorp.com/terraform/language/modules/sources

**Question: 80** CertyIQ

You write a new Terraform configuration and immediately run terraform apply in the CLI using the local backend. Why will the apply fail?

A. Terraform needs you to format your code according to best practices first
B. Terraform needs to install the necessary plugins first
C. The Terraform CLI needs you to log into Terraform cloud first
D. Terraform requires you to manually run terraform plan first

**Answer: B**

**Explanation:**

The correct answer is **B. Terraform needs to install the necessary plugins first.** Here's why:

Terraform relies on plugins, called providers, to interact with various infrastructure platforms and services (e.g., AWS, Azure, Google Cloud, Docker). These providers are specific to the resource types you define in your configuration. When a new Terraform configuration is created, the necessary providers are not immediately

available. Terraform needs to download these plugins based on the required_providers block specified in your configuration or implicitly identified by the resources being used. Before terraform apply can execute, Terraform needs to examine your configuration, identify the required providers, and download them to the local machine. If these plugins are missing, the terraform apply command will fail because it won't know how to communicate with the underlying infrastructure services to provision the specified resources. This process is essential for ensuring that Terraform has the correct tooling to manage your infrastructure.Option A is incorrect because while formatting best practices improve readability and maintainability, they do not hinder the fundamental process of plan and apply. Option C is wrong because using Terraform cloud is optional and not mandatory to run a basic terraform plan and apply workflow with the local backend. Option D is incorrect because while running plan first is a best practice and recommended for users to review the changes, it is not mandatory and one can directly run apply, which automatically generates the plan and the apply the plan.

For further understanding, you can refer to the official Terraform documentation:

**Terraform Providers:** https://www.terraform.io/docs/providers/index.html
**Required Providers:** https://www.terraform.io/docs/language/providers/required-providers.html
**Terraform init:** https://www.terraform.io/docs/cli/commands/init.html (This command is the one that installs the plugins)

# Thank you

Thank you for being so interested in the premium exam material.
I'm glad to hear that you found it informative and helpful.

## But Wait

I wanted to let you know that there is more content available in the full version.
The full paper contains additional sections and information that you may find helpful, and I encourage you to download it to get a more comprehensive and detailed view of all the subject matter.