#1 Terraform Associate Certification 003 - Results

Back to result overview

Attempt 1

All domains

67 all
37 correct
28 incorrect
2 skipped
0 marked

Collapse all questions

Question 1 **Incorrect**

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

split

Correct answer

slice

Explanation

Slice is a collection function, all other answers are string functions.

Your answer is incorrect

chomp

join

Overall explanation

Answer: slice

Explanation:

In Terraform, the `slice` function is not specifically a string function; rather, it is used to work with lists (arrays) by extracting a subset of elements from them.

Here's a breakdown of each option:

- **split**: Valid string function. It splits a string into a list of substrings based on a specified delimiter.
  - Example: `split(",", "a,b,c")` results in `["a", "b", "c"]`.
- **join**: Valid string function. It concatenates a list of strings into a single string with a specified separator.
  - Example: `join(",", ["a", "b", "c"])` results in `"a,b,c"`.
- **slice**: Not a valid string function. It is used to create a new list from a given list by specifying a range of indices.
  - Example: `slice(["a", "b", "c", "d"], 1, 3)` results in `["b", "c"]`.
- **chomp**: Valid string function. It removes trailing newline characters from a string.
  - Example: `chomp("hello\n")` results in `"hello"`.

Resources

Terraform Built-in Functions

Question 2 **Correct**

A provider configuration block is required in every Terraform configuration.

Example:

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

True

Explanation

This statement is incorrect. A provider configuration block is not required in every Terraform configuration. Provider configurations are used to define the specific cloud provider or service that Terraform will interact with, but they are not mandatory for every

configuration. In some cases, Terraform can automatically discover providers, or the provider configuration can be inherited from other modules.

Your answer is correct

False

Explanation

This statement is correct. A provider configuration block is not required in every Terraform configuration. While provider configurations are essential for defining the cloud provider or service that Terraform will interact with, they are not mandatory for all configurations. Terraform configurations can include resources, data sources, variables, and other elements without necessarily needing a provider configuration.

Overall explanation

A provider configuration block is **not required** in every Terraform configuration. If a module is being used that already specifies the provider, or if the provider is configured elsewhere (like in a parent module or in Terraform Cloud's workspace settings), then it's not necessary to include the provider block in your configuration. Terraform can also automatically find and install required providers from the registry if they are referenced.

However, in most cases, a provider block is necessary to define how to interact with external APIs and manage resources (like AWS, Azure, or Google Cloud) unless you are explicitly inheriting the provider configuration from a parent module.

Resources

Terraform Providers

Terraform Provider Configuration

Question 3Incorrect

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

Correct answer

The ability to restrict modules to members of Terraform Cloud or Enterprise organizations

Explanation

The private module registry in Terraform Cloud/Terraform Enterprise provides the ability to restrict modules to members of Terraform Cloud or Enterprise organizations. This means that only authorized users within the organization can access and use the modules, ensuring that sensitive infrastructure code is not exposed to unauthorized users.

The ability to tag modules by version or release

Explanation

Tagging modules by version or release is a feature available in both the public Terraform Public Module Registry and the private module registry in Terraform Cloud/Terraform Enterprise. This feature allows users to easily identify and manage different versions of modules, regardless of whether they are public or private.

Your answer is incorrect

The ability to share modules with public Terraform users and members of Terraform Enterprise Organizations

Explanation

The private module registry in Terraform Cloud/Terraform Enterprise does not provide the ability to share modules with public Terraform users. It is specifically designed to restrict access to modules to members of Terraform Cloud or Enterprise organizations, ensuring a higher level of control and security over the modules.

The ability to share modules publicly with any user of Terraform

Explanation

Sharing modules publicly with any user of Terraform is a feature of the public Terraform Module Registry, not the private module registry in Terraform Cloud/Terraform Enterprise. The private registry is designed for internal use within organizations, while the public registry is available for sharing modules with the broader Terraform community.

Overall explanation

Answer: The ability to restrict modules to members of Terraform Cloud or Enterprise organizations

Explanation:

The private module registry in Terraform Cloud and Terraform Enterprise allows organizations to share modules internally while restricting access to only authorized members of the organization. This ensures that sensitive or proprietary modules are not publicly accessible, providing better control over who can use or modify those modules.

Incorrect Options:

- **The ability to share modules with public Terraform users and members of Terraform Enterprise Organizations**: This option describes the public registry, which is accessible to everyone, not a private registry.
- **The ability to tag modules by version or release**: Both public and private registries allow versioning of modules; this is not a differentiating feature.
- **The ability to share modules publicly with any user of Terraform**: This is a feature of the public Terraform Module Registry, not the private registry.

Resources
Terraform Private Modules

Question 4Incorrect
Which option can **not** be used to keep secrets out of Terraform configuration files?

A -var flag
Explanation
The -var flag in Terraform allows users to pass variables from the command line when running Terraform commands. While it can be used to pass sensitive information, it is not the most secure method for keeping secrets out of Terraform configuration files.

Environment variables
Explanation
Environment variables are commonly used to store sensitive information such as API keys, passwords, and tokens outside of Terraform configuration files. They provide a way to pass secret values to Terraform without exposing them in the code.

Your answer is incorrect
A Terraform provider
Explanation
A Terraform provider is used to interact with APIs, services, and resources. While it can be used to securely access external resources, it is not specifically designed to keep secrets out of Terraform configuration files.

Correct answer
secure string
Explanation
Secure strings, or encrypted strings, are not a native feature in Terraform for keeping secrets out of configuration files. Terraform does not have built-in support for securely storing and managing sensitive information like passwords or API keys.

Overall explanation
Answer: secure string

Explanation:

Terraform does not have a built-in `secure string` option specifically for keeping secrets out of configuration files. Here's an overview of the other options and why they can be used to keep secrets out of Terraform configuration files:

- **A Terraform provider**: Some providers, such as AWS or Vault, allow you to manage secrets securely, either by fetching them from a secure location or by managing secrets in a dedicated secrets management service.
- **Environment variables**: Environment variables can be used to pass sensitive data (like API keys or passwords) without including them directly in configuration files. Terraform can read these using variable definitions.
- **A `-var` flag**: This command-line flag allows you to specify sensitive variable values when you run Terraform commands, so they don't need to be stored in the configuration file.

Resources
Protect sensitive input variables

Question 5Incorrect
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"
   }
}
```

aws_ami.ubuntu

Explanation

The value "aws_ami.ubuntu" is not a valid reference to the AMI data source in Terraform. It does not provide the necessary information to retrieve the specific AMI ID required for the AWS instance resource.

Your answer is incorrect

aws_ami.ubuntu.id

Explanation

The value "aws_ami.ubuntu.id" is not a valid reference to the AMI data source in Terraform.

data.aws_ami.ubuntu

Explanation

The value "data.aws_ami.ubuntu" is a reference to the data source itself, but it does not provide the specific attribute needed to retrieve the AMI ID required for the AWS instance resource.

Correct answer

data.aws_ami.ubuntu.id

Explanation

The value "data.aws_ami.ubuntu.id" is the correct reference to the AMI ID attribute of the data source. This value will provide the specific AMI ID required for the AWS instance resource to use in the configuration.

Overall explanation

Answer: `data.aws_ami.ubuntu.id`

Explanation:

To reference the ID of an AMI (Amazon Machine Image) from a `data` source in Terraform, you use the syntax `data.<data_source>.<name>.<attribute>`. In this case, the correct reference for the AMI ID is `data.aws_ami.ubuntu.id`.

Each part of this syntax means:

- `data` signifies that it's a data source.
- `aws_ami` is the type of data source.
- `ubuntu` is the name given to this data source in the configuration.
- `id` accesses the specific attribute (the AMI ID) that's required for the `ami` argument in the instance resource.

Using `data.aws_ami.ubuntu.id` will correctly populate the AMI ID when creating the AWS instance.

Resources
aws_instance Terraform Resource

Question 6 Correct
Terraform requires the **Go** runtime as a prerequisite for installation.

Your answer is correct
False
Explanation
This choice is correct because Terraform does not require the Go runtime as a prerequisite for installation. Users can install Terraform directly without needing to install Go separately.

True
Explanation
Terraform is written in the Go programming language, but it does not require the Go runtime as a prerequisite for installation. Terraform can be installed on various operating systems without the need for installing Go separately.

Overall explanation
Answer: False

Explanation:

Terraform does not require the Go runtime as a prerequisite for installation. Terraform is a standalone binary that can be installed and run independently without needing Go or any other runtime environment. You can download the appropriate binary for your operating system from the official Terraform website and install it without any additional dependencies.

Resources
Terraform Installation Guide

Question 7 Correct
What is the provider for this fictitious resource?

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

vpc
Your answer is correct
aws
Explanation
In the resource block `resource "aws_vpc" "main"`, the provider is identified by the prefix before the resource type, which is `aws`. This indicates that the `aws` provider is responsible for managing this resource, in this case, an Amazon VPC (Virtual Private Cloud).

test
main
Overall explanation
Answer: aws

Explanation:

In this example, the **provider** is "aws." The provider is specified in the resource block prefix (`aws_vpc`), where "aws" indicates the cloud provider or service used to manage this resource.

Here's a breakdown of each option:

- **vpc**: Incorrect. "vpc" is part of the resource type (`aws_vpc`), but not the provider. It indicates the type of resource being created within the AWS provider.

- **main**: Incorrect. "main" is the **resource name** or label given to the instance of the `aws_vpc` resource, which is used to reference this specific VPC in other parts of the configuration.
- **aws**: Correct. "aws" is the provider, which is responsible for managing resources in Amazon Web Services (AWS).
- **test**: Incorrect. "test" is a value assigned to the `name` attribute, which is a specific property of the VPC resource, not the provider.

Resources
AWS Provider for Terraform

Question 8 **Correct**
What command does Terraform require the first time you run it within a configuration directory?

terraform import
Explanation
The `terraform import` command is used to import existing infrastructure into Terraform. It is not required the first time you run Terraform within a configuration directory, as the initial setup and configuration are done with the `terraform init` command.

terraform workspace
Explanation
The `terraform workspace` command is used to manage workspaces in Terraform, allowing you to switch between different states and configurations. It is not required the first time you run Terraform within a configuration directory, as the initial setup is done with the `terraform init` command.

Your answer is correct
terraform init
Explanation
The `terraform init` command is required the first time you run Terraform within a configuration directory. It initializes the working directory and downloads any necessary plugins and modules to set up the environment for Terraform to run.

terraform plan
Explanation
The `terraform plan` command is used to create an execution plan showing what Terraform will do when you run `terraform apply`. While it is an important command in the Terraform workflow, it is not required the first time you run Terraform within a configuration directory.

Overall explanation
The `terraform init` command is required the first time you run Terraform in a new configuration directory. This command initializes the working directory by:

1. Downloading the necessary provider plugins.
2. Setting up the backend for storing state (if configured).
3. Preparing the configuration for Terraform to run.

- `terraform import` is used to import existing resources into Terraform's state but isn't required on the first run.
- `terraform plan` is used to preview infrastructure changes, but it requires the directory to be initialized first with `terraform init`.
- `terraform workspace` manages workspaces for multiple environments but also requires initialization first.

Resources
Terraform Init

Question 9 **Incorrect**
Which provisioner invokes a process on the resource created by Terraform?

file
Explanation
The file provisioner is used to copy files or directories from the machine running Terraform to the resource created by Terraform. It is not used to invoke a process on the resource, but rather to transfer files or configurations to the provisioned resource.

Your answer is incorrect
local-exec
Explanation
The local-exec provisioner is used to execute commands locally on the machine running Terraform, not on the resource created by Terraform. It is commonly used for tasks like running scripts or commands on the local machine during the Terraform execution, but it does not invoke a process on the provisioned resource.

null-exec

Explanation

The null-exec provisioner is used to do nothing when invoked. It is typically used when a resource needs to be created without any additional actions or commands executed on it. It does not invoke any process on the resource created by Terraform, so it is not the correct choice for this scenario.

Correct answer

remote-exec

Explanation

The remote-exec provisioner is used to invoke a process on the resource created by Terraform. It allows running commands on the remote resource after it has been created, making it a suitable choice for executing scripts or commands on the provisioned resource.

Overall explanation

The `remote-exec` provisioner is used to run commands on the remote machine after it has been created by Terraform. It is commonly used with resources like servers where you need to execute commands (like shell scripts) on the created resource over SSH or WinRM.

- `null-exec` is not a valid Terraform provisioner.
- `local-exec` runs commands on the **local machine** where Terraform is executed, not on the resource created by Terraform.
- `file` is used to upload files to the remote resource but doesn't execute processes.

Resources

Remote-Exec Provisioner

Question 10 **Incorrect**

What is not processed when running a terraform refresh?

Correct answer

Configuration file

Explanation

The **configuration file** is not processed when running a terraform refresh. The refresh command is used to reconcile the state of the infrastructure with the actual resources in the cloud provider, so it does not need to reprocess the configuration file.

Cloud provider

Explanation

The cloud provider is processed when running a terraform refresh. The refresh command communicates with the cloud provider to gather the latest state of the resources and update the Terraform state file accordingly.

State file

Explanation

The state file is processed when running a terraform refresh because it is used to compare the current state of the infrastructure with the state declared in the configuration files. This helps Terraform determine what changes need to be applied to bring the infrastructure to the desired state.

Your answer is incorrect

Credentials

Explanation

Credentials are not directly processed when running a terraform refresh. Credentials are typically used to authenticate Terraform with the cloud provider, but they are not reprocessed during a refresh operation.

Overall explanation

Answer: Configuration file

Explanation:

When running `terraform refresh`, Terraform only updates the **state file** to match the current state of resources in the cloud provider. It does not re-process or re-evaluate the **configuration file** (the `.tf` files containing your resource definitions).

Here's a breakdown of each option:

- **State file**: Incorrect. The state file is processed and updated to reflect the actual state of the resources.
- **Configuration file**: Correct. The configuration file is not re-evaluated during `terraform refresh`; it only updates the state with information from the cloud provider.
- **Credentials**: Incorrect. Credentials are used to authenticate with the cloud provider and access the current resource states.
- **Cloud provider**: Incorrect. Terraform communicates with the cloud provider to check the current status of resources and updates the state file accordingly.

Resources
Terraform refresh command

Question 11 **Incorrect**
You should store secret data in the same version control repository as your Terraform configuration.

Correct answer
False

Explanation
It is not recommended to store secret data such as passwords, API keys, or other sensitive information in your version control system (VCS). This is a general best practice in software development, not just with Terraform.

Your answer is incorrect
True

Overall explanation
Answer: False

Explanation:

You should **not** store secret data in the same version control repository as your Terraform configuration. Storing sensitive information, such as passwords or API keys, in a public or shared repository poses a security risk. Instead, best practices recommend using secure storage solutions for secrets, such as:

- **Environment variables**: Store secrets as environment variables on the machine where Terraform is run.
- **Terraform Vault provider**: Use HashiCorp Vault to manage and access secrets securely.
- **Encrypted files**: Use tools to encrypt secret files and manage their access appropriately.

By keeping sensitive data separate from your version-controlled code, you can enhance the security of your infrastructure.

Resources
Sensitive Data in State

Question 12 **Incorrect**
What does the default "**local**" Terraform backend store?

Terraform binary

Explanation
The Terraform binary is the executable file used to run Terraform commands and manage infrastructure. The default "local" Terraform backend does not store the Terraform binary itself.

Provider plugins

Explanation
Provider plugins are external software components that extend Terraform's capabilities to interact with different cloud providers and services. The default "local" Terraform backend does not store provider plugins.

Correct answer
State file

Explanation
The default "local" Terraform backend stores the state file, which contains information about the current state of the infrastructure managed by Terraform. This file is crucial for tracking changes, managing resources, and ensuring consistency across deployments.

Your answer is incorrect
tfplan files

Explanation

tfplan files are temporary files generated by Terraform during the planning phase to show the changes that will be applied to the infrastructure. These files are not stored in the default "local" Terraform backend.

Overall explanation
Answer: State file

Explanation:

The default "local" Terraform backend stores the Terraform state file, which contains information about the resources that Terraform manages. This state file is essential for Terraform to track the current state of the infrastructure, allowing it to perform operations like `apply`, `destroy`, and `plan` effectively.

Incorrect Options:

- **tfplan files**: `tfplan` files are temporary files created during the planning phase when using the `terraform plan` command. They are not stored by the backend.
- **Terraform binary**: The Terraform binary is the executable file used to run Terraform commands, but it is not stored by the backend.
- **Provider plugins**: Provider plugins are binaries that Terraform uses to interact with various cloud providers and are typically stored in a separate directory (like `.terraform/plugins`) but not in the backend.

Resources
Terraform Local Backend

Question 13 **Incorrect**
Setting the **TF_LOG** environment variable to DEBUG causes debug messages to be logged into syslog.

Correct answer
False
Explanation
This choice is correct because setting the TF_LOG environment variable to DEBUG does not result in debug messages being logged into syslog. The TF_LOG environment variable is specific to Terraform and controls the verbosity of Terraform's logging output, not the logging mechanism itself.

Your answer is incorrect
True
Explanation
Setting the TF_LOG environment variable to DEBUG does not cause debug messages to be logged into syslog. Instead, it enables debug-level logging for Terraform itself, allowing users to see detailed information about Terraform operations, resource changes, and errors in the console or log files.

Overall explanation
Answer: False

Explanation:

Setting the `TF_LOG` environment variable to `DEBUG` causes detailed debug messages to appear on **stderr** (standard error output), not in **syslog**. Terraform's logging system outputs to the terminal by default, not to system logs like syslog.

You can adjust the verbosity by setting `TF_LOG` to levels like `TRACE`, `DEBUG`, `INFO`, `WARN`, or `ERROR`, or use `JSON` for machine-readable logs at the TRACE level or higher.

Resources
Debugging Terraform

Question 14 **Correct**
How does Terraform store its state in a backend?

Only in an S3 bucket
Explanation

While Terraform can store its state in an S3 bucket when configured to do so, it is not the only option for backend storage. Terraform supports various backend configurations, including remote storage solutions like Terraform Cloud, Azure Storage, Google Cloud Storage, and more.

Locally on the developer's machine

Explanation

Storing Terraform state locally on the developer's machine is not recommended for production environments as it can lead to issues with state management, collaboration, and consistency across team members.

Your answer is correct

In a remote location defined by the backend configuration

Explanation

Terraform stores its state in a backend, which is a remote location defined by the backend configuration. This allows for better state management, collaboration, and consistency among team members working on the same infrastructure.

Only in Terraform Cloud

Explanation

Terraform can store its state in Terraform Cloud, but it is not the only option. Terraform supports various backend configurations, and Terraform Cloud is just one of the remote storage solutions available for storing state.

Overall explanation

Terraform can store its state in various backends, including remote options like S3, Terraform Cloud, Azure Blob Storage, and more. The backend configuration in your `terraform` code defines where this state is stored. You can use both local and remote backends based on the setup. Terraform automatically manages the state to keep track of the resources it creates, updates, or deletes.

- Options "**Locally on the developer's machine**" and "**Only in an S3 bucket**" are incorrect because Terraform supports multiple backends beyond just local storage and S3.
- Option "**Only in Terraform Cloud**" is incorrect as it's just one of the many remote backend options available.

Resources

Terraform state Backends

Question 15Correct

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

Your answer is correct

True

Explanation

In Terraform 0.13 and above, outside of the required_providers block, Terraform configurations always refer to providers by their local names. This means that when referencing a provider in a resource block, data block, or module block, you should use the local name defined in the provider block, rather than the full provider source address. This change was introduced to simplify provider management and make configurations more concise and readable.

False

Explanation

False. In Terraform 0.13 and above, outside of the required_providers block, Terraform configurations always refer to providers by their local names. This change was made to improve the readability and maintainability of Terraform configurations by using local provider names instead of full provider source addresses.

Overall explanation

True

In Terraform 0.13 and above, configurations reference providers by their local names outside of the `required_providers` block. The `required_providers` block is where the source and version for each provider are defined, but within other parts of the configuration, you use the local names specified in that block to refer to the providers.

For example:

```
terraform {
 required_providers {
```

```
    aws = {
      source  = "hashicorp/aws"
      version = "~> 4.0"
    }
  }
}

provider "aws" {
  region = "us-west-2"
}

resource "aws_instance" "example" {
  ami           = "ami-123456"
  instance_type = "t2.micro"
}
```

Here, `aws` is the local name used in the configuration to refer to the AWS provider specified in the `required_providers` block.

Resources

Local Names

Question 16 **Correct**

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

Your answer is correct

All of the above

Explanation

The public Terraform Module Registry automatically exposes information about published modules, including required **input variables**, **optional input variables** and their **default values**, and **outputs**. This information is displayed on the module's page in the registry, which can be accessed by anyone.

Required input variables

None of the above

Outputs

Optional inputs variables and default values

Overall explanation

Answer: All of the above

Explanation:

The public Terraform Module Registry automatically exposes information about the following aspects of published modules:

- **Required input variables**: The registry displays input variables that must be provided for the module to function.
- **Optional input variables and default values**: Optional variables are also listed, along with their default values if specified.
- **Outputs**: The registry shows the outputs that the module will produce, helping users understand what values they can access after applying the module.

This visibility makes it easier for users to understand and configure modules correctly.

Resources

AWS VPC Public Module

Question 17 **Correct**

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

Your answer is correct

False

Explanation

This choice is correct. While many standard backend types support state storage, locking, and remote operations, not all of them do. It is essential to be aware of the specific capabilities of each backend type when working with Terraform to ensure that the necessary functionalities are available for managing infrastructure.

True

Explanation

This choice is incorrect. Not all standard backend types support state storage, locking, and remote operations like plan, apply, and destroy. Some backend types may not have the capability to perform these operations, making this statement false.

Overall explanation

False

Not all standard backend types support state locking and remote operations like plan, apply, and destroy. While many remote backends, such as Terraform Cloud, S3 with DynamoDB, or Azure Blob Storage with state locking, support these features, local backends or certain other backends (like some third-party storage options) might not support state locking or remote operations. The level of functionality depends on the backend configuration.

Resources

Backend block configuration overview

Question 18 **Incorrect**

When should you use the `force-unlock` command?

You have a high priority change

Explanation

Having a high priority change does not automatically warrant the use of the force-unlock command. It is important to follow best practices and only resort to force-unlock when necessary, such as in cases of automatic unlocking failure.

Correct answer

Automatic unlocking failed

Explanation

The force-unlock command should be used when automatic unlocking has failed. This command allows you to manually release a stuck lock on a state file, ensuring that you can continue with necessary operations without being blocked by a lock that cannot be automatically released.

Your answer is incorrect

You apply failed due to a state lock

Explanation

If a apply failed due to a state lock, using the force-unlock command may not be the appropriate solution. It is important to first investigate the cause of the state lock and attempt to resolve it through standard procedures before resorting to force-unlock as a last resort.

You see a status message that you cannot acquire the lock

Explanation

Using the force-unlock command is not necessary when you see a status message that you cannot acquire the lock.

Overall explanation

Answer: Automatic unlocking failed

Explanation:

The `terraform force-unlock` command is used when the automatic unlocking of a state lock fails. Terraform uses state locking to prevent concurrent operations that could lead to inconsistencies in the state file. If a process that held the lock crashes or fails unexpectedly, the lock may remain in place. In such cases, you can use `terraform force-unlock` to manually remove the lock and allow subsequent operations to proceed.

Incorrect Options:

- **You see a status message that you cannot acquire the lock**: This indicates that the lock is held by another process. You should only use `force-unlock` if you are sure the lock is stale and cannot be automatically released.
- **You have a high priority change**: While you may have high priority changes, using `force-unlock` without ensuring the lock is stale could lead to state corruption.
- **You apply failed due to a state lock**: If an apply fails due to a state lock, the best practice is to check why the lock is held and ensure that the process holding the lock is no longer running before using `force-unlock`.

Resources

Terraform Force Unlock

Question 19 Incorrect

Terraform providers are always installed from the Internet.

Your answer is incorrect

True

Correct answer

False

Overall explanation

Terraform providers are not always installed from the internet. They can be sourced from multiple locations, including local file paths, a network mirror, or a private registry. This flexibility allows for offline installations or using custom providers in specific environments.

Resources

Provider Installation

Question 20 Correct

`terraform validate` validates the syntax of Terraform files.

False

Explanation

This choice is incorrect.

Your answer is correct

True

Explanation

This choice is correct because the `terraform validate` command is used to validate the syntax of Terraform files. It checks the configuration files for syntax errors, ensuring that they are written correctly and can be successfully processed by Terraform.

Overall explanation

Answer: True

Explanation:

The `terraform validate` command is used to check the syntax and validity of the Terraform configuration files in your project. It ensures that the files are well-formed and adhere to Terraform's requirements but does not check the actual infrastructure or state. This command is useful for catching errors in your Terraform code before you attempt to apply or plan any changes.

Resources

Command: validate

Question 21 Correct

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

Your answer is correct

Run terraform refresh

Explanation

The command had been deprecated already. Use `-refresh-only` option along with `terraform plan` or `terraform apply` for the same operation.

It will happen automatically

Manually update the state file

Run terraform import

Overall explanation

Answer: Run `terraform refresh`

Explanation:

If infrastructure is manually destroyed, the best practice to update Terraform's understanding of the current state is to run `terraform refresh`. This command refreshes the state file, allowing Terraform to detect changes and accurately reflect the current infrastructure status.

Here's an explanation of each option:

- **Run `terraform refresh`**: Correct. `terraform refresh` updates Terraform's state file with the real state of resources, enabling it to recognize that the infrastructure has been destroyed.
- **It will happen automatically**: Incorrect. Terraform does not automatically detect changes made outside of its control. Manual intervention, like running `terraform refresh` or `terraform plan`, is necessary to reflect these changes.
- **Manually update the state file**: Incorrect. Directly editing the state file is risky and not recommended as it may corrupt the state. `terraform refresh` is safer and will update the state accurately.
- **Run `terraform import`**: Incorrect. `terraform import` is used to bring existing resources under Terraform management, not to reflect deletions. Since the resource was deleted, this command would not be appropriate.

Resources
Terraform Command: refresh

Question 22 **Incorrect**
What features does the hosted service Terraform Cloud provide? **(Choose two.)**

Your selection is correct
A web-based user interface (UI)
Explanation
A web-based user interface (UI) is a feature provided by Terraform Cloud. The UI allows users to manage their infrastructure, view state information, collaborate with team members, and access various features of Terraform Cloud through a user-friendly interface.

Automated infrastructure deployment visualization
Explanation
Automated infrastructure deployment visualization is not a feature provided by Terraform Cloud. While Terraform Cloud offers various features to manage infrastructure as code, visualization of infrastructure deployment is not one of them.

Correct selection
Remote state storage
Explanation
Remote state storage is a key feature provided by Terraform Cloud. It allows users to store their Terraform state files securely in the cloud, enabling collaboration, version control, and easy access to state data from multiple locations.

Your selection is incorrect
Automatic backups
Explanation
Automatic backups are not a feature provided by Terraform Cloud. Terraform Cloud focuses on managing infrastructure as code and collaboration among team members, rather than providing backup services for infrastructure resources.

Overall explanation
Answer: Remote state storage & A web-based user interface (UI)

Explanation:

Terraform Cloud offers several features to enhance the infrastructure management experience. Among these, the following are key features:

- **Remote state storage**: Terraform Cloud provides a secure remote backend for storing your Terraform state files. This feature helps teams manage state files centrally and enables collaboration without the risks associated with local state files.
- **A web-based user interface (UI)**: Terraform Cloud includes a user-friendly web interface that allows users to manage their infrastructure, view workspace details, trigger runs, and monitor the status of their Terraform operations visually.

Incorect Options:

- **Automated infrastructure deployment visualization**: While Terraform Cloud provides some visualization of the infrastructure, the term "automated infrastructure deployment visualization" is more associated with tools like Terraform Enterprise or integrations rather than a core feature of Terraform Cloud.
- **Automatic backups**: Terraform Cloud does not specifically provide a feature labeled "automatic backups." Users are advised to manage their state files with appropriate backup strategies, which may involve using remote state storage features.

Resources
HCP Terraform Plans and Features

Question 23 Correct

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

True

Explanation

This statement is incorrect. Terraform can run on both Windows and Linux operating systems, and it does not specifically require a Server version of the Windows operating system.

Your answer is correct

False

Explanation

This statement is correct. Terraform is compatible with both Windows and Linux operating systems, and it does not mandate the use of a Server version of the Windows operating system. Users can choose the operating system that best fits their needs and preferences when running Terraform.

Overall explanation

Answer: False

Explanation:

Terraform can run on **both** Windows and Linux operating systems, but it does not specifically require a Server version of Windows.

Terraform is compatible with various versions of Windows, including Windows 10, Windows 11, and Windows Server editions. Users can download and run Terraform on these platforms without the need for a particular Windows Server version.

Resources

Install Terraform

Question 24 Correct

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

Your answer is correct

None of the above

Explanation

None of the above arguments are required when declaring a Terraform variable. Variables can be declared without specifying a data type, default value, or description. However, it is good practice to include these arguments for better code organization and understanding.

type

description

default

Explanation

The 'default' argument is not required when declaring a Terraform variable. It is used to provide a default value for the variable if no value is explicitly set, but it is not mandatory to include in the variable declaration.

All of the above

Overall explanation

Answer: None of the above

Explanation:

In Terraform, when declaring a variable, none of the arguments (`type`, `default`, `description`) are strictly required. You can declare a variable with just its name, and it will be valid. However, these arguments can be added to enhance the variable's clarity and usability:

- `type`: Optional. It specifies the expected type of the variable (e.g., string, number, list, map).
- `default`: Optional. It sets a default value for the variable, which is used if no value is provided when running Terraform.
- `description`: Optional. It provides a description of the variable for documentation purposes but does not affect its functionality.

Example:

Here's a simple variable declaration without any arguments:

```
variable "my_variable" {}
```

Resources

Declaring an Input Variable

Question 25 **Correct**

What is one disadvantage of using dynamic blocks in Terraform?

Your answer is correct

They make configuration harder to read and understand

Explanation

One disadvantage of using dynamic blocks in Terraform is that they can make the configuration harder to read and understand. Since dynamic blocks introduce a level of abstraction and flexibility, it may be challenging for someone unfamiliar with the codebase to quickly grasp the structure and logic of the configuration.

Terraform will run more slowly

Explanation

The use of dynamic blocks in Terraform does not directly impact the speed at which Terraform runs. While complex configurations may introduce some overhead, the performance impact of dynamic blocks themselves on Terraform's execution speed is minimal.

They cannot be used to loop through a list of values

Explanation

Dynamic blocks in Terraform are actually designed to loop through a list of values, so this statement is incorrect.

Dynamic blocks can construct repeatable nested blocks

Explanation

Dynamic blocks are capable of constructing repeatable nested blocks, allowing for the creation of complex and dynamic configurations. This feature is one of the advantages of using dynamic blocks in Terraform, not a disadvantage.

Overall explanation

Answer: They make configuration harder to read and understand

Explanation:

Dynamic blocks in Terraform are powerful for creating repeatable nested blocks programmatically, but this flexibility can come at the cost of readability. Here's an explanation of each option:

- **They cannot be used to loop through a list of values**: This is incorrect because dynamic blocks are specifically designed to loop through lists of values to generate nested blocks dynamically.
- **Dynamic blocks can construct repeatable nested blocks**: This is a true statement, but it describes a benefit of dynamic blocks, not a disadvantage.
- **They make configuration harder to read and understand**: This is correct. Using dynamic blocks can make the code less readable, especially for users who are not familiar with their syntax or who prefer explicit declarations. It can be harder to follow the logic and structure of the configuration compared to using straightforward, static definitions.
- **Terraform will run more slowly**: This is not true; dynamic blocks do not inherently slow down Terraform. The execution time is generally unaffected by the use of dynamic blocks.

Resources

Terraform Best Practice for dynamic Blocks

Question 26 **Correct**

Which of the following is not true of Terraform providers?

Some providers are maintained by HashiCorp

Explanation

Some Terraform providers are indeed maintained by HashiCorp, the company behind Terraform. These officially maintained providers are typically well-documented and supported by the Terraform community.

Your answer is correct

None of the above

Explanation

The statement "None of the above" is not true because all of the previous statements are accurate and describe various aspects of Terraform providers accurately.

Major cloud vendors and non-cloud vendors can write, maintain, or collaborate on Terraform providers

Explanation

Major cloud vendors like AWS, Azure, and Google Cloud, as well as non-cloud vendors, can write, maintain, or collaborate on Terraform providers. This allows for a wide range of providers to be available for different services and platforms.

Providers can be maintained by a community of users

Explanation

Terraform providers can be maintained by a community of users, enabling collaborative efforts to improve and update existing providers or create new ones to meet specific needs or integrate with different services.

Providers can be written by individuals

Explanation

Providers in Terraform can indeed be written by individuals, allowing for a wide range of community-contributed providers to extend Terraform's capabilities beyond what is officially supported.

Overall explanation

The correct answer is None of the above.

Explanation:

All of the provided statements about Terraform providers are true:

- Individuals can write and maintain their own Terraform providers to support specific APIs or services.
- Providers can be maintained by a community of users. Many open-source providers are collaboratively maintained.
- HashiCorp maintains some of the most widely used providers, including the AWS and Azure providers.
- Major cloud vendors (e.g., AWS, Google Cloud, Microsoft Azure) and non-cloud vendors (e.g., GitHub, Datadog) can write and collaborate on Terraform providers to integrate their services with Terraform.

Since all statements are true, **None of the above** is the correct choice.

Resources

Terraform Provider Registry

Terraform Providers Documentation

Question 27Correct

What is `terraform refresh` intended to detect?

Your answer is correct

State file drift

Explanation

The correct choice. Terraform refresh is intended to detect state file drift, which occurs when the actual infrastructure drifts away from the state described in the Terraform state file. It helps Terraform to update its state to reflect the current state of the infrastructure.

Corrupt state files

Explanation

Terraform refresh is not primarily focused on detecting corrupt state files. It is used to synchronize the Terraform state with the actual infrastructure state to ensure accurate and up-to-date information for future Terraform operations.

Empty state files

Explanation

Terraform refresh is not specifically designed to detect empty state files. It is used to update the state file with the current state of the infrastructure.

Terraform configuration code changes

Explanation

Terraform refresh is not intended to detect changes in the Terraform configuration code itself. It is used to reconcile the Terraform state with the real-world infrastructure.

Overall explanation

State file drift

The `terraform refresh` command is intended to detect drift between the Terraform state file and the actual infrastructure by updating the state file with the latest resource statuses from the infrastructure. This helps ensure that Terraform's representation of

the infrastructure is accurate and up to date before applying any changes.

Warning: This command is deprecated, because its default behavior is unsafe if you have misconfigured credentials for any of your providers.

Resources
Command: refresh

Question 28 **Correct**
You have deployed a new **web app** 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?

Run `terraform output ip_address` to view the result

Explanation
Running `terraform output ip_address` will only work if you have explicitly defined an output variable named `ip_address` in your Terraform configuration. Since you did not create any outputs for your code, this command will not provide the IP address of the deployed resource.

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 in the state file

Explanation
Using the `terraform_remote_state` data source to load in the state file and writing outputs for each resource is a valid approach for accessing information from a separate state file. However, in this scenario where you did not create any outputs, this method would require additional configuration and is not the most efficient way to quickly find the IP address of the deployed resource.

Run `terraform destroy` then `terraform apply` and look for the IP address in stdout

Explanation
Running `terraform destroy` followed by `terraform apply` is not a recommended approach to find the IP address of a deployed resource. This method involves destroying and recreating the infrastructure, which is unnecessary and time-consuming. Additionally, looking for the IP address in the stdout output may not be reliable or efficient.

Your answer is correct

Run `terraform state list` to find the name of the resource, then `terraform state show` to find the attributes including the public IP address

Explanation
Running `terraform state list` will provide you with a list of all the resources managed by Terraform. By then running `terraform state show`, you can view the attributes of a specific resource, including the public IP address. This method allows you to quickly find the IP address without the need to create additional outputs or modify your Terraform configuration.

Overall explanation
Explanation:
The best approach to quickly find the IP address of a deployed resource is to use the Terraform state commands. By running `terraform state list`, you can identify all the resources managed by Terraform in your state file. Once you know the name of the resource (e.g., the web app), you can run `terraform state show <resource_name>` to see the details of that resource, including attributes like the public IP address.

- **"Run `terraform output ip_address` to view the result"** is incorrect because `terraform output` will not work if you did not create any outputs.
- **"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 in the state file"** involves more steps and is less efficient for quickly finding an IP address, as it requires setting up a new directory and writing outputs.
- **"Run `terraform destroy` then `terraform apply` and look for the IP address in stdout"** is not recommended since destroying and recreating resources is disruptive and unnecessary for retrieving the IP address.

Resources
Terraform state list command

Question 29 Correct

Why would you use the `terraform taint` command?

When you want Terraform to destroy all the infrastructure in your workspace

When you want to force Terraform to destroy a resource on the next apply

When you want Terraform to ignore a resource on the next apply

Your answer is correct

When you want to force Terraform to destroy and recreate a resource on the next apply

Explanation

Utilizing the `terraform taint` command is beneficial when you specifically want Terraform to destroy and recreate a particular resource during the **next apply**. This can be crucial when you need to ensure that the resource is updated with new configurations or settings.

Warning: This command is deprecated. For Terraform v0.15.2 and later, Terraform recommend using the `-replace` option with `terraform apply` instead

Overall explanation

Answer: When you want to force Terraform to destroy and recreate a resource on the next apply

Explanation:

The `terraform taint` command marks a resource for recreation during the next `terraform apply`. When you taint a resource, Terraform will understand that the current state of that resource is not valid or needs to be updated. As a result, the resource will be destroyed and then recreated on the next apply.

Warning: The `terraform taint` command is deprecated. For Terraform versions 0.15.2 and later, it is recommended to use the `-replace` option with the `terraform apply` command instead. This change improves clarity by specifying which resources to replace directly in the apply command.
Check here: Taint Command

Incorrect Options:

- **When you want to force Terraform to destroy a resource on the next apply**: This is incorrect because tainting does not just destroy a resource; it also recreates it.
- **When you want Terraform to ignore a resource on the next apply**: This is incorrect because tainting a resource explicitly indicates that it should be replaced, not ignored.
- **When you want Terraform to destroy all the infrastructure in your workspace**: This is incorrect as `terraform taint` only affects specific resources. To destroy all infrastructure, you would use the `terraform destroy` command.

Resources

Command: taint

Terraform Replace

Question 30 Correct

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

True

Your answer is correct

False

Overall explanation

The **`terraform.tfstate`** file does *not* always match your currently built infrastructure. While it reflects the state of resources managed by Terraform at the time of the last `apply`, manual changes made directly in the cloud provider or through other tools can lead to a state "drift." Terraform does not automatically track these external changes, which means that the state file can become out of sync with the actual infrastructure.

Example of Drift

For instance, if you manually delete an AWS EC2 instance that Terraform created, the `terraform.tfstate` file will still show that instance as existing until you run a `terraform plan` or `terraform apply`, which will then indicate that there is a difference (drift).

Terraform state File: https://developer.hashicorp.com/terraform/language/state
Manage Resource drift: https://developer.hashicorp.com/terraform/tutorials/state/resource-drift
Resources
Terraform state File Documentation

Question 31 **Incorrect**
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}"
}
```

Correct answer
aws_instance.web[1].name
Explanation
In Terraform, when referencing elements in a list, the index starts at 0. Therefore, aws_instance.web[1].name would correctly reference the "name" value of the second instance in the list of aws_instance.web.
aws_instance.web.*.name
Explanation
The syntax aws_instance.web.*.name is used in Terraform to reference the "name" attribute of all instances of the aws_instance.web resource. It does not specify a specific instance, so it would not be the correct way to reference the "name" value of the second instance specifically.
Your answer is incorrect
aws_instance.web[2].name
Explanation
In Terraform, when accessing elements in a list, the index starts at 0. Therefore, aws_instance.web[2].name would actually reference the third instance, not the second. To reference the second instance, the correct index would be 1.
aws_instance.web[1]
Explanation
In Terraform, when accessing elements in a list, the index starts at 0. Therefore, aws_instance.web[1] would actually reference the second instance, not the third. However, to access the "name" value, you need to include ".name" at the end.
element(aws_instance.web, 2)
Explanation
The element() function in Terraform is used to access a specific element in a list or map. In this case, the correct index for the second instance would be 1, not 2, as indexing starts at 0. Therefore, element(aws_instance.web, 2) would actually reference the third instance, not the second.
Overall explanation
aws_instance.web[1].name

In Terraform, when using the `count` parameter, each instance of a resource can be accessed by its index, starting from zero. To reference the "name" value of the second instance, use `aws_instance.web[1].name`, as this accesses the second instance (index 1) directly.
Resources
Referring to Instances

Question 32 **Incorrect**

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

Your answer is incorrect

True

Correct answer

False

Explanation

The statement is accurate. When executing the terraform taint command on a managed resource in Terraform, it does not result in an instant destruction and recreation of the resource. The resource will only be recreated during the following apply process after being tainted.

Overall explanation

False

When you run `terraform taint` on a managed resource, it marks the resource as "tainted," indicating that it should be recreated during the next `terraform apply`. However, the resource will not be immediately destroyed and recreated when you run `terraform taint`. It will only be destroyed and recreated during the subsequent `terraform apply`.

Resources

Command: taint

Question 33Correct

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

The datasource block

Explanation

The datasource block is used to fetch data from external sources or services that can be used in your Terraform configurations. It is not where you would specify the state backend configuration, as the state backend is a fundamental setting that determines how Terraform manages and stores the state of your infrastructure.

The provider block

Explanation

The provider block is used to configure the provider plugins that Terraform uses to interact with specific cloud platforms or services (e.g., AWS, Azure). While the provider block is essential for defining the connection to your target infrastructure, it is not where you would specify the state backend configuration.

The resource block

Explanation

The resource block is used to define and configure individual resources within your infrastructure, such as virtual machines, databases, or networks. It is not where you would specify the state backend configuration, as the state backend is a higher-level configuration setting for managing the state of your Terraform deployments.

Your answer is correct

The terraform block

Explanation

The state backend configuration is specified within the terraform block in your Terraform configuration. This is where you define settings such as the type of state backend (e.g., local, remote), backend-specific configurations (e.g., storage account for remote backend), and other related options.

Overall explanation

Answer: The `terraform` block

Explanation:

In Terraform, you specify a state backend within the `terraform` block. This block configures how and where Terraform stores the state file, which records information about the managed infrastructure. By defining a backend, you can control where the state file is stored (e.g., locally or remotely in services like AWS S3, Terraform Cloud, or HashiCorp Consul).

Example:

Here's an example of how to configure an S3 backend in the `terraform` block:

```
terraform {
 backend "s3" {
   bucket = "my-terraform-state-bucket"
   key    = "path/to/my/terraform.tfstate"
   region = "us-west-2"
 }
}
```

In this example, the state file will be stored in an S3 bucket in the `us-west-2` region, at the specified path within the bucket.

Resources

Define a backend block

Question 34 **Correct**

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?

Your answer is correct

Run the `terraform fmt` command during the code linting phase of your CI/CD process

Explanation

Running the `terraform fmt` command during the code linting phase of your CI/CD process is the correct approach to format Terraform HCL code according to standard Terraform style convention. This command automatically formats the code based on the official Terraform style guide, ensuring consistency and readability across all team members' code.

Designate one person in each team to review and format everyone's code

Explanation

Designating one person in each team to review and format everyone's code may introduce inconsistencies in the formatting of Terraform HCL code. It is more efficient and reliable to use automated tools like `terraform fmt` to enforce standard formatting conventions.

Write a shell script to transform Terraform files using tools such as AWK, Python, and sed

Explanation

Writing a shell script to transform Terraform files using tools such as AWK, Python, and sed may introduce complexity and potential errors in the formatting process. It is more straightforward and reliable to utilize the built-in `terraform fmt` command for consistent and standardized formatting of Terraform HCL code.

Manually apply two spaces indentation and align equal sign "=" characters in every Terraform file (*.tf)

Explanation

Manually applying two spaces indentation and aligning equal sign "=" characters in every Terraform file (*.tf) is a time-consuming and error-prone process. It is not the recommended approach for maintaining consistent formatting standards across multiple team members working on infrastructure as code using Terraform.

Overall explanation

Answer: Run the terraform fmt command during the code linting phase of your CI/CD process

Explanation:

The `terraform fmt` command automatically formats Terraform configuration files (HCL) according to standard Terraform style conventions, ensuring consistent indentation and alignment. Integrating this command into the CI/CD pipeline during the code linting phase helps maintain code quality and readability across team members.

Incorrect Options:

- **Designate one person in each team to review and format everyone's code**: This approach is not efficient or scalable and may lead to inconsistencies, as it relies on manual reviews rather than automated formatting.
- **Manually apply two spaces indentation and align equal sign "=" characters in every Terraform file (*.tf)**: Manually formatting code is error-prone and tedious; it's better to use an automated tool.

- **Write a shell script to transform Terraform files using tools such as AWK, Python, and sed**: While this is technically possible, it's unnecessary because `terraform fmt` already provides a reliable solution for formatting.

Resources

"terraform fmt" Command

Question 35 **Correct**

What is the workflow for deploying new infrastructure with Terraform?

Your answer is correct

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

Explanation

The correct workflow for deploying new infrastructure with Terraform involves writing a Terraform configuration, initializing the Terraform environment with terraform init, planning the infrastructure changes with terraform plan, and applying those changes with terraform apply. This workflow ensures that changes are properly planned and executed in a controlled manner.

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

Explanation

Using the terraform plan command to import the current infrastructure to the state file is not the correct workflow for deploying new infrastructure with Terraform. The plan command is used to preview the changes that Terraform will make, not to import existing infrastructure.

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

Explanation

The workflow described in this choice, using terraform import to import the current infrastructure to the state file, is not the correct approach for deploying new infrastructure with Terraform. The import command is used to bring existing infrastructure under Terraform management, not for creating new infrastructure.

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

Explanation

Running terraform show to view proposed changes is not part of the workflow for deploying new infrastructure with Terraform. The show command is used to inspect the current state as recorded in the Terraform state file, not to view proposed changes.

Overall explanation

Explanation:

The correct workflow for deploying new infrastructure using Terraform is:

1. Write a Terraform configuration that defines the desired infrastructure resources.
2. Run `terraform init` to initialize the working directory, set up the backend (if using remote), and download any provider plugins.
3. Run `terraform plan` to preview the changes that Terraform will make to the infrastructure based on the configuration.
4. Run `terraform apply` to actually create or update the infrastructure as defined in the configuration.

- **"terraform plan to import the current infrastructure to the state file, make code changes, and terraform apply to update the infrastructure."** is incorrect because `terraform plan` does not import the current infrastructure; `terraform plan` is used to generate a plan to create or update resources.
- **"Write a Terraform configuration, run terraform show to view proposed changes, and terraform apply to create new infrastructure."** is incorrect because `terraform show` is used to display the state, not preview changes.
- **"terraform import to import the current infrastructure to the state file, make code changes, and terraform apply to update the infrastructure."** is incorrect because `terraform import` is only used to bring existing infrastructure under Terraform management but isn't part of the standard workflow for deploying new infrastructure.

Resources

Terraform Workflow

Question 36 **Correct**

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

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

azurerm_resource_group

Explanation

"azurerm_resource_group" is not the name assigned by Terraform to reference a specific resource. This choice represents a resource type within the Azure provider in Terraform, specifically a resource group. It is not the specific name assigned to reference the resource.

Your answer is correct

dev

Explanation

In Terraform, the name assigned to reference a specific resource is typically defined by the user within the Terraform configuration file. In this case, "dev" is a valid and commonly used name that can be assigned to reference the resource in question.

azurerm

Explanation

"azurerm" is not the name assigned by Terraform to reference a specific resource. This choice represents the Azure provider in Terraform, which is used to interact with Azure resources. It is not the specific name assigned to reference the resource in question.

test

Overall explanation

Answer: dev

Explanation:

In Terraform, each resource is referenced by a specific resource type and a user-defined name. In this example:

- `azurerm_resource_group` is the **resource type**, indicating that this is an Azure resource group.
- `"dev"` is the **name assigned by Terraform to reference this specific resource**. This identifier is unique within the configuration file and is used to reference this resource in other parts of the Terraform configuration.

Thus, to refer to this resource elsewhere in the configuration, you would use `azurerm_resource_group.dev`.

Question 37Correct

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

Your answer is correct

False

Explanation

No, variable descriptions are not stored in Terraform's state file

True

Overall explanation

Answer: B. False

Explanation:

Terraform variables and outputs that use the "description" argument **do not store** this description in the state file. The description is purely for documentation purposes in the configuration files and helps users understand the purpose of variables and outputs. The state file only stores the values of these variables and outputs, not their descriptions.

Resources

Terraform State Documentation

Question 38Correct

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

True

Explanation

This statement is incorrect. The `terraform init` command initializes a Terraform working directory containing Terraform configuration files. It does not automatically create a sample `main.tf` file in the current directory.

Your answer is correct

False

Explanation

This statement is correct. The `terraform init` command initializes a Terraform working directory by downloading provider plugins and modules specified in the configuration files. It does not create a sample `main.tf` file in the current directory.

Overall explanation

Answer: False

Explanation:

The `terraform init` command initializes a Terraform working directory containing configuration files, but it does not create a sample `main.tf` file. Instead, it sets up the backend, installs the necessary provider plugins, and prepares the environment for further Terraform commands.

If a `main.tf` file or other configuration files are not already present in the directory, you need to create those manually.

Resources

Command: init

Question 39Correct

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?

terraform apply -target=null_resource.run_script

Explanation

Using the 'terraform apply -target=null_resource.run_script' command directly may not be the most efficient approach in this situation. The 'terraform taint' command is specifically designed to mark a resource as tainted and trigger its recreation, which is more appropriate for rerunning the local-exec provisioner script.

Your answer is correct

terraform taint null_resource.run_script

Explanation

The correct command to use first in this scenario is 'terraform taint null_resource.run_script'. This command marks the null resource as tainted, causing it to be recreated during the next 'terraform apply' operation. This will ensure that the local-exec provisioner script is rerun as needed.

terraform plan -target=null_resource.run_script

Explanation

The 'terraform plan -target=null_resource.run_script' command is used to generate an execution plan for the specified target resource, but it does not trigger the rerun of the local-exec provisioner script in the null resource. The 'terraform taint' command is more suitable for marking the resource as tainted and ensuring the script is rerun.

terraform validate null_resource.run_script

Explanation

The 'terraform validate' command is used to validate the configuration files for syntax errors and other issues, but it does not have the functionality to rerun a local-exec provisioner script in a null resource. Therefore, it is not the correct command to use in this scenario.

Overall explanation

In Terraform versions prior to v0.15.2, the `terraform taint` command was used to mark a resource for recreation during the next apply. In this case, `terraform taint null_resource.run_script` would mark the `null_resource.run_script` for recreation, allowing the local-exec provisioner to rerun the script when you apply the changes.

However, **in Terraform v0.15.2 and later**, the `terraform taint` command has been deprecated. Instead, you should use the `terraform apply` or `terraform plan` command with the `-replace` option to force replacement of the object:

```
terraform apply -replace=null_resource.run_script
```

- `terraform apply -target=null_resource.run_script` would only apply changes to the specific resource but wouldn't mark it for recreation unless something had changed.
- `terraform validate null_resource.run_script` is used for validating configuration files and doesn't trigger any resource changes.
- `terraform plan -target=null_resource.run_script` shows what Terraform will do but doesn't mark the resource for recreation.

Resources
Terraform Taint Command (deprecated)
Terraform Replace

Question 40 **Correct**
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.)**

Run the terraform import-gcp command
Explanation
Option D is also incorrect because there is no terraform import-gcp command in Terraform. The correct command for importing GCP resources is simply terraform import.

Provision new VMs using Terraform with the same VM names
Explanation
Option A is incorrect because provisioning new VMs with the same names would create duplicate resources and could cause conflicts with the existing VMs.

Your selection is correct
Use the terraform import command for the existing VMs
Your selection is correct
Write Terraform configuration for the existing VMs
Overall explanation
Answer: Use the `terraform import` command for the existing VMs
Answer: Write Terraform configuration for the existing VMs

Explanation:

To manage existing VMs on Google Cloud Platform (GCP) with Terraform, you need to follow these steps:

- **Use the `terraform import` command for the existing VMs**: This command allows you to bring the existing resources under Terraform management by associating them with Terraform state. You'll need the resource identifier for each VM to use this command effectively.
- **Write Terraform configuration for the existing VMs**: You need to create a Terraform configuration file that defines the existing VMs as resources. This configuration should match the existing setup in GCP so that Terraform can manage them accurately.

Incorrect Options:

- **Provision new VMs using Terraform with the same VM names**: While using the same names might help with organization, it is not a requirement. Importing the existing VMs allows you to manage them without needing to provision new ones.
- **Run the `terraform import-gcp` command**: There is no specific command called `terraform import-gcp`. The correct command is simply `terraform import`, which can be used for any provider, including GCP.

Resources
Terraform Import

Question 41 Correct

Terraform provisioners can be added to any resource block.

Your answer is correct

True

Explanation

True. Terraform provisioners can indeed be added to any resource block within a Terraform configuration file. Provisioners allow you to run scripts or commands on a resource after it is created, which can be useful for tasks like initializing the resource, configuring it, or performing additional setup steps.

False

Explanation

False. This statement is incorrect as Terraform provisioners can be added to any resource block within a Terraform configuration file. Provisioners provide a way to execute scripts or commands on a resource after it is created, allowing for additional customization and setup.

Overall explanation

True

Terraform provisioners can be added to any resource block. They are used to execute scripts or commands on the created resources, such as installing software or configuring resources after they are created.

Example:

```
resource "aws_instance" "web" {
 # ...

 provisioner "local-exec" {
   command = "echo The server's IP address is ${self.private_ip}"
 }
}
```

However, it's generally recommended to use provisioners sparingly as they can lead to non-idempotent operations and make your infrastructure less predictable.

Resources

How to use Provisioners

Question 42 Correct

"Terraform can import modules from a number of sources "

Which of the following is NOT a valid source?

Local path

Explanation

Local paths are a valid source for importing modules in Terraform. Users can specify the local path to a directory containing the module files in their Terraform configuration.

Your answer is correct

FTP server

Explanation

Terraform does not support importing modules directly from an FTP server.

GitHub repository

Explanation

GitHub repositories are a valid source for importing modules in Terraform. Users can specify the GitHub repository URL in their Terraform configuration to import modules from GitHub.

Terraform Module Registry

Explanation

The Terraform Module Registry is a central repository where users can publish, discover, and share Terraform modules. It is a valid source for importing modules in Terraform and provides a convenient way to access and use pre-built modules for infrastructure provisioning.

Overall explanation

Answer: FTP server

Explanation:

Terraform can import modules from various sources, including:

- **GitHub repository**: Modules can be sourced directly from GitHub repositories by specifying the repository URL.
- **Local path**: You can reference modules located on your local filesystem by providing a relative or absolute path.
- **Terraform Module Registry**: The Terraform Module Registry is a central repository for publicly available Terraform modules, and you can directly reference modules hosted there.

However, **FTP server** is not a valid source for importing Terraform modules. Terraform does not natively support sourcing modules from FTP servers.

Resources

Module Sources

Question 43Incorrect

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

Correct answer

True

Explanation

Yes you can expose Local values through Outputs in Terraform

Your answer is incorrect

False

Overall explanation

Answer: A. True

Explanation:

In Terraform, if a module defines a **local** value, you can expose that value by referencing it in a `terraform output`. This is often used to make useful information accessible outside the module, like a public IP address.

Example

Suppose a module has a local value for the public IP of an instance:

```
locals {
 instance_public_ip = aws_instance.example.public_ip
}

output "public_ip" {
 value = local.instance_public_ip
}
```

In this example:

- The `instance_public_ip` local value stores the public IP of an AWS instance.
- The `output` block then exposes `instance_public_ip` outside the module, making it accessible in the root module or for other resources referencing this module.

Resources

Terraform Locals

Terraform Output Values

Question 44Incorrect

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

Correct answer

False

Explanation

1 workspace -> 1 backend

1 backend -> multiple workspaces

Your answer is incorrect

True

Explanation

The remote backend can work with either a single remote HCP Terraform workspace, or with multiple similarly-named remote workspaces

Overall explanation

A single remote backend can support multiple workspaces. This allows you to have different environments (like `dev`, `stage`, `prod`) in separate workspaces, all stored within the same backend.

So, **1 backend can support multiple workspaces**, but each workspace only uses one backend at a time.

Resources

Backend

Question 45Correct

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.)**

Your selection is correct

Run `terraform plan -destroy`.

Explanation

Running `terraform plan -destroy` will show you a detailed execution plan of what Terraform will do when destroying the infrastructure. It will display all the resources that will be deleted, allowing you to review the changes before applying them.

Your selection is correct

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

Explanation

By running `terraform destroy`, Terraform will output a plan of all the resources that will be deleted before actually destroying them. This allows you to review the changes and ensure that only the intended resources are being deleted. It is a safe practice to confirm the deletion of resources before proceeding with the destruction.

Run `terraform state rm *`.

Explanation

Running `terraform state rm *` is not the correct command to show all the resources that will be deleted. This command is used to remove a resource from the Terraform state, not to preview the resources that will be deleted during a destroy operation.

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

Explanation

This statement is incorrect. Terraform provides the functionality to show resources that will be deleted by using commands like `terraform plan -destroy` or `terraform destroy`.

Overall explanation

Answer: Run `terraform plan -destroy`.

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

Explanation:

- **Run `terraform plan -destroy`.**: This command shows a detailed plan of what will be destroyed without actually performing the destruction. It lists all the resources that Terraform will remove, allowing you to review them before proceeding.

- **Run `terraform destroy` and it will first output all the resources that will be deleted before prompting for approval.**: When you execute `terraform destroy`, Terraform will show you the resources that it plans to delete and ask for confirmation before proceeding. This is a safe way to ensure you know exactly what will be removed.

Incorrect Options:

- **This is not possible. You can only show resources that will be created.**: This statement is incorrect because Terraform provides commands to view resources that will be destroyed, as mentioned above.
- **Run `terraform state rm *`.**: This command is used to remove items from the Terraform state file, but it does not provide information about what will be destroyed. In fact, using this command incorrectly can lead to orphaned resources.

Resources
Command: plan

Question 46 **Incorrect**
Which of the following is **not** a valid Terraform collection type?

Correct answer
tree
Explanation
Tree is not a valid Terraform collection type. Terraform does not have a built-in tree data structure for organizing data. Instead, Terraform uses lists, maps, and sets to manage collections of data.

list
Explanation
In Terraform, a list is a collection type that allows you to store multiple values in a specific order. Lists are commonly used to represent arrays of values in Terraform configurations.

Your answer is incorrect
map
Explanation
In Terraform, a map is a collection type that allows you to store key-value pairs. Maps are commonly used to represent configurations with named attributes and values.

set
Explanation
In Terraform, a set is a collection type that allows you to store unique values. Sets are commonly used to ensure uniqueness in a collection of values and to perform set operations like union, intersection, and difference.

Overall explanation
tree

In Terraform, the valid collection types are:

- **list**: An ordered sequence of values.
- **map**: A collection of key-value pairs where each key is unique.
- **set**: An unordered collection of unique values.

"**tree**" is not a valid collection type in Terraform.

Resources
Collection Types

Question 47 **Correct**
Only the user that generated a plan may apply it.

True
Explanation
This statement is incorrect. In Terraform, the plan and apply steps are separate actions that can be performed by different users. The user who generated the plan is not the only one who can apply it. Multiple users with the necessary permissions can apply the plan generated by another user.

Your answer is correct
False

Explanation
This statement is correct. In Terraform, the user who generated a plan is not the only one who can apply it. Other users with the appropriate permissions can also apply the plan, making the statement "Only the user that generated a plan may apply it" false.
Overall explanation
Answer: False

Explanation:

Terraform does not restrict the application of a generated plan to the user who created it. Any user with the necessary permissions can apply a plan file created by someone else. This flexibility allows team members to collaborate on infrastructure management, as one user can generate the plan, and another with appropriate permissions can apply it.

While Terraform permits this flexibility, it's considered best practice to manage permissions carefully, especially in production environments, to ensure changes are reviewed and approved by authorized personnel before application.

Question 48 Correct
A Terraform local value can reference other Terraform local values.
Your answer is correct
True
Explanation
True. In Terraform, local values can reference other local values within the same module. This allows for creating reusable and modular code by defining values once and using them in multiple places within the configuration.
False
Explanation
False. In Terraform, local values can indeed reference other local values. This feature enables better organization and reusability of code by allowing values to be defined and used within the same module.
Overall explanation
A Terraform local value can reference other Terraform local values. This allows you to create complex expressions by combining multiple local values.

For **example**:

```
locals {
 value1 = "Hello"
 value2 = "World"
 combined_value = "${local.value1} ${local.value2}"
}
```

In this example, `combined_value` references both `value1` and `value2`.
Resources
Declaring a Local Value

Question 49 Skipped
Please fill the blank field(s) in the statement with the right words.

You need to specify a dependency manually.
__ 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.
Correct answer
depends_on
Explanation
Answer: `depends_on`

Explanation:

The `depends_on` meta-parameter is used in Terraform to specify explicit dependencies between resources. By using `depends_on`, you ensure that a resource waits until other specified resources are created, updated, or deleted before it proceeds. This is helpful when Terraform cannot automatically infer dependencies and allows you to control the resource provisioning order directly.

Resources

The depends_on Meta-Argument

Question 50 Correct

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?

Your answer is correct

Add version = "1.0.0" attribute to module block

Explanation

Adding version = "1.0.0" attribute to the module block is the correct way to specify the version when referencing a module stored on the public Terraform Module Registry. This ensures that the module is pulled at the specified version for consistency and predictability.

Append ?ref=v1.0.0 argument to the source path

Explanation

Appending ?ref=v1.0.0 to the source path is not the correct way to specify a version when using a module block to reference a module stored on the public Terraform Module Registry. The correct method involves using the version attribute within the module block.

Nothing - modules stored on the public Terraform Module Registry always default to version 1.0.0

Explanation

Modules stored on the public Terraform Module Registry do not default to version 1.0.0 automatically. It is important to explicitly specify the version when referencing a module to ensure that the desired version is used in the configuration.

Modules stored on the public Terraform Module Registry do not support versioning

Explanation

Modules stored on the public Terraform Module Registry do support versioning. It is possible to specify a specific version when referencing a module to ensure consistency and compatibility.

Overall explanation

Answer: Add `version = "1.0.0"` attribute to module block

Explanation:

To specify a version for a module sourced from the public Terraform Module Registry, you can include a `version` argument within the module block. This ensures that you are using the correct version of the module you want to work with.

Here's how you would correctly reference version 1.0.0 of the Consul module:

```
module "consul" {
  source  = "hashicorp/consul/aws"
  version = "1.0.0"
}
```

Incorrect Options:

- **Modules stored on the public Terraform Module Registry do not support versioning**: This is incorrect. Modules can be versioned.
- **Append `?ref=v1.0.0` argument to the source path**: This is not how versioning is specified for modules in the Terraform Module Registry. The correct approach is to use the `version` argument.
- **Nothing — modules stored on the public Terraform Module Registry always default to version 1.0.0**: This is also incorrect. If no version is specified, Terraform will use the latest available version.

Resources

Terraform documentation to define Module Version

Question 51 **Incorrect**

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

Your answer is incorrect

terraform show workspace

Explanation

The command "terraform show workspace" is not a valid Terraform command. It combines "terraform show" and "terraform workspace" commands, which are used for displaying resource state and managing workspaces separately. This combination would not provide the desired outcome of listing all workspaces for the current configuration.

terraform workspace show

Explanation

The command "terraform workspace show" is used to display the current workspace, not all workspaces for the current configuration. It provides information about the current workspace, such as its name and associated variables, but does not list all available workspaces.

Correct answer

terraform workspace list

Explanation

The command "terraform workspace list" is the correct choice as it is specifically used to display all workspaces for the current configuration. It lists all available workspaces, allowing you to see the names of each workspace and easily switch between them.

terraform workspace

Explanation

The command "terraform workspace" is used to manage Terraform workspaces, but it does not specifically display all workspaces for the current configuration. It is used for switching, creating, deleting, and listing workspaces, but not specifically for displaying all workspaces.

Overall explanation

The `terraform workspace list` command displays all available workspaces for the current configuration. This command shows the current workspace and other existing workspaces, allowing you to manage and switch between them as needed.

Resources

Command: workspace list

Question 52 **Incorrect**

A Terraform provider is not responsible for:

Your answer is incorrect

Exposing resources and data sources based on an API

Explanation

Exposing resources and data sources based on an API is a responsibility of a Terraform provider. Providers define the resources and data sources that can be managed within Terraform configurations, allowing users to interact with them through the provider's API.

Understanding API interactions with some service

Explanation

Understanding API interactions with a service is not the responsibility of a Terraform provider. The provider is responsible for defining and managing the resources and data sources within the infrastructure, not for understanding the underlying API interactions.

Correct answer

Provisioning infrastructure in multiple clouds

Explanation

Provisioning infrastructure in multiple clouds is not the responsibility of a Terraform provider. While Terraform itself can be used to provision infrastructure across multiple clouds, this task is handled by the Terraform configuration and not by the provider.

Managing actions to take based on resource differences

Explanation

Managing actions to take based on resource differences is a responsibility of a Terraform provider. Providers are responsible for detecting changes in the infrastructure configuration and applying the necessary actions to reconcile those differences, ensuring that the desired state is maintained.

Overall explanation

Provisioning infrastructure in multiple clouds

A Terraform provider is responsible for understanding the API interactions with a specific service, exposing resources and data sources based on an API, and managing actions to take based on resource differences.

However, it is not specifically responsible for provisioning across multiple clouds; instead, each provider is typically designed to interact with a particular cloud or service. Terraform can manage multi-cloud deployments by using multiple providers, but each provider itself is usually dedicated to a single service or platform.

Resources

Providers

Question 53 **Correct**

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**?

Implement a checklist that engineers can follow when configuring compute instances

Explanation

Implementing a checklist can help standardize the configuration process for compute instances, but it does not automate the deployment process using infrastructure as code. It may help reduce human error but does not provide a scalable and automated solution.

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

Explanation

Implementing a ticketing workflow may help track manual provisioning and configuration tasks, but it does not address the root cause of inconsistent configurations and does not automate the deployment process using infrastructure as code.

Your answer is correct

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

Explanation

Implementing a provisioning pipeline that deploys infrastructure configurations committed to a version control system following code reviews is the correct choice. This approach automates the deployment process using infrastructure as code, ensuring consistent configurations across compute instances and reducing the risk of human error. It enables scalable and repeatable deployments, improving reliability and efficiency.

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

Explanation

Replacing the compute instance type with a larger version may address performance issues, but it does not solve the problem of manual provisioning and configuration leading to inconsistent configurations. It does not leverage the benefits of infrastructure as code for automated and consistent deployments.

Overall explanation

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

Explanation:

To solve the problem of inconsistent configurations between compute instances, using Infrastructure as Code (IaC) is the best approach. Implementing a provisioning pipeline allows for:

- **Consistency**: Infrastructure configurations defined in code ensure that all instances are provisioned with the same settings and configurations.
- **Version Control**: Storing your configurations in a version control system (like Git) allows for better tracking of changes, collaboration, and rollback capabilities.
- **Automated Deployment**: A provisioning pipeline can automate the deployment process, reducing the risk of human error associated with manual deployments and configurations.
- **Code Reviews**: Including code reviews in the workflow ensures that configurations are evaluated by peers before being applied, which can lead to better practices and early detection of issues.

Incorrect Options:

- **Implement a ticketing workflow that makes engineers submit a ticket before manually provisioning and configuring a resource**: While this may add some control, it does not address the root cause of inconsistency and relies on manual processes.
- **Implement a checklist that engineers can follow when configuring compute instances**: A checklist may help improve manual processes, but it is still prone to human error and does not ensure consistent application.
- **Replace the compute instance type with a larger version to reduce the number of required deployments**: This does not solve the problem of inconsistent configurations; it merely addresses the symptoms by scaling up the resources.

Resources
Infrastructure as Code: What Is It? Why Is It Important?

Question 54 Incorrect
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.)**

Correct selection
var.list[*].id
Explanation
This choice correctly uses the [*] syntax to extract the id attribute from each object in the list. It will produce a list of all the IDs from the objects in var.list.
{ for o in var.list : o => o.id }
Explanation
This choice uses a for loop to iterate over each object in the list and create a map where the key is the object itself and the value is the id attribute of the object. It does not directly produce a list of IDs as requested in the question.

Your selection is incorrect
[var.list[*].id]
Explanation
This choice uses the [*] syntax to extract the id attribute from each object in the list, but it wraps the result in square brackets, creating a list of lists. This is not the correct syntax to produce a flat list of IDs.

Correct selection
[for o in var.list : o.id]
Explanation
This choice uses a for loop to iterate over each object in the list and retrieve the id attribute from each object. It accurately creates a list containing all the IDs from the objects in var.list.

Overall explanation
Correct Answers:

- `var.list[*].id`: This syntax uses the splat operator (`*`) to extract the `id` attribute from each object in the list, producing a flat list of IDs.
- `[ for o in var.list : o.id ]`: This is a for-expression that iterates over each object `o` in `var.list` and constructs a new list containing the `id` attribute of each object.

Incorrect Options:

- `{ for o in var.list : o => o.id }`: This is a map expression, not a list. It would produce a map with keys as the entire object `o` and values as `o.id`, which is not what is required.
- `[ var.list[*].id ]`: This will create a list containing a single element, which is itself a list of IDs, resulting in a nested list rather than a flat list of IDs.

Resources

Terraform Splat Expressions

Terraform For Expressions

Question 55 **Correct**

Where does the Terraform local backend store its state?

In the terraform file

Explanation

SThe terraform file is used to define the infrastructure configuration, not to store the state information.

In the /tmp directory

Explanation

The /tmp directory is typically used for temporary files and may not provide the necessary persistence and reliability for storing the state information.

In the user's terraform.state file

Explanation

There is no concept of a "user's terraform.state file" in Terraform. The state file is typically named terraform.tfstate and is used to store the state information for the infrastructure managed by Terraform.

Your answer is correct

In the terraform.tfstate file

Explanation

The Terraform local backend stores its state in the **terraform.tfstate** file by default. This file contains the current state of the infrastructure as managed by Terraform.

Overall explanation

Answer: In the terraform.tfstate file

Explanation:

The Terraform local backend stores its state in a file named `terraform.tfstate` located in the current working directory where the Terraform commands are run. This file contains all the information about the resources that Terraform manages and their current state.

Incorrect answers:

- **In the /tmp directory**: This is incorrect. The local backend does not automatically store the state in the `/tmp` directory; it uses the current working directory by default.
- **In the terraform file**: This is misleading. The term "terraform file" is not specific enough and does not accurately describe where the state is stored.
- **In the user's terraform.state file**: While this sounds similar, the default state file name is `terraform.tfstate`, not `terraform.state`.

Resources

Terraform Local Backend

Question 56 **Incorrect**

Please fill the blank field(s) in the statement with the right words.

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.

Your answer is incorrect
terrafrom.tfstate

Correct answer
terraform.tfstate

Explanation
The 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.

Resources
Terraform State

Question 57 **Incorrect**
A Terraform **provisioner** must be nested inside a resource configuration block.

Correct answer
True

Explanation
True. In Terraform, provisioners are used to execute scripts or commands on a resource after it is created. To ensure that the provisioner runs in the correct context and order, it must be nested inside the resource configuration block where the resource is defined. Placing the provisioner outside the resource block will result in an error and the provisioner will not be executed as intended.

Your answer is incorrect
False

Explanation
False. In Terraform, provisioners are specifically designed to be nested inside a resource configuration block.

Overall explanation
Answer: True

Explanation:

In Terraform, provisioners are used to execute scripts or commands on a resource after it has been created. For a provisioner to work correctly, it must be defined within the context of a specific resource configuration block. This ensures that the provisioner knows which resource it is acting upon and when to run its commands.

Example:

```
resource "aws_instance" "example" {
 ami           = "ami-12345678"
 instance_type = "t2.micro"

 provisioner "local-exec" {
   command = "echo Instance created: ${self.public_ip}"
 }
}
```

In this example, the `local-exec` provisioner is nested inside the `aws_instance` resource block, allowing it to execute the command once the instance is created.

Resources
Terraform Provisioners

Question 58 **Incorrect**
When does **terraform apply** reflect changes in the cloud environment?

Immediately

Explanation
Terraform apply does not reflect changes immediately in the cloud environment. It sends the request to the resource provider and waits for the provider to fulfill the request, which can take some time depending on the provider's speed and workload.

Based on the value provided to the -refresh command line argument

Explanation

The -refresh command line argument in terraform apply is used to update the state file with the current state of the infrastructure before making any changes. It does not directly control when the changes are reflected in the cloud environment.

Correct answer

However long it takes the resource provider to fulfill the request

Explanation

This choice is correct because terraform apply reflects changes in the cloud environment based on how long it takes the resource provider to fulfill the request. The time taken can vary depending on the complexity of the changes, the provider's responsiveness, and other factors.

Your answer is incorrect

None of the above

Explanation

None of the above choices accurately describe when terraform apply reflects changes in the cloud environment. The correct timing is when the resource provider fulfills the request, which can vary in duration.

After updating the state file

Explanation

After updating the state file, terraform apply does not automatically reflect changes in the cloud environment. The state file is used to track the current state of the infrastructure and is updated after the apply operation is completed.

Overall explanation

Correct Answer: **However long it takes the resource provider to fulfill the request**

The `terraform apply` command sends requests to the cloud provider to create, update, or delete resources according to the configuration. Changes are reflected in the cloud environment **based on the time required by the cloud provider** to fulfill those requests, which varies depending on the resource and provider.

Resources

Apply Terraform configuration

Question 59Correct

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

It is only available to paying customers

Explanation

The availability of the Terraform remote backend is not limited to paying customers only. It is a feature that is accessible to all users of Terraform, regardless of their subscription or payment status.

It doesn't show the output of a terraform apply locally

Explanation

This statement is not directly related to the difference between the Terraform remote backend and other state backends. The ability to show the output of a `terraform apply` locally is a feature of Terraform itself and is not specific to the type of backend being used.

All of the above

Explanation

This choice is incorrect because not all the options listed are true. The Terraform remote backend's ability to execute runs on dedicated infrastructure and its availability to all users are valid points, but the statement about showing the output of `terraform apply` locally and the requirement for paying customers are not accurate representations of the differences between the remote backend and other state backends.

Your answer is correct

It can execute Terraform runs on dedicated infrastructure on premises or in Terraform Cloud

Explanation

The Terraform remote backend allows users to store their state file remotely on dedicated infrastructure, either on premises or in Terraform Cloud. This differs from other state backends like S3 or Consul, which may not provide the same level of control over where the state file is stored and accessed.

Overall explanation

Backends define where Terraform's state snapshots are stored.

A given Terraform configuration can either specify a backend, integrate with Terraform Cloud, or do neither and default to storing state locally.

The **Terraform remote backend** stores state remotely, enabling collaboration. Unlike backends like **S3** or **Consul** (which only store state), the remote backend can also trigger and execute Terraform runs. In **Terraform Cloud**, these runs happen on dedicated infrastructure managed by HashiCorp, while **Terraform Enterprise** allows execution on on-premises infrastructure. This ensures secure, scalable, and collaborative infrastructure management.

- **It doesn't show the output of a terraform apply locally** is incorrect because The output of terraform apply can still be viewed locally even with a remote backend.
- **It is only available to paying customers** is incorrect option: While certain features of Terraform Cloud require a paid subscription, the remote backend itself is available in both free and paid versions.

Resources
Backend block configuration overview

Question 60 **Incorrect**
Which of these is the best practice to protect sensitive values in state files?

Secure Sockets Layer (SSL)
Explanation
Secure Sockets Layer (SSL) is a protocol used to secure communication over a network, such as between a client and a server. While SSL can help secure data in transit, it is not specifically designed to protect sensitive values in Terraform state files.

Blockchain
Explanation
Blockchain technology is not typically used to protect sensitive values in state files. While blockchain can provide secure and immutable data storage, it is not the most practical or commonly used method for protecting sensitive values in Terraform state files.

Correct answer
Enhanced remote backends
Explanation
Enhanced remote backends, such as using encrypted storage solutions like AWS S3 with server-side encryption or Azure Blob Storage with encryption at rest, are considered best practices for protecting sensitive values in Terraform state files. These backends provide secure storage for state files and help prevent unauthorized access to sensitive information.

Your answer is incorrect
Signed Terraform providers
Explanation
Signed Terraform providers are not directly related to protecting sensitive values in state files. Signed providers help verify the authenticity and integrity of Terraform provider plugins but do not specifically address the protection of sensitive values in state files.

Overall explanation
Enhanced remote backends

Enhanced remote backends, such as Terraform Cloud, Terraform Enterprise, or cloud storage solutions (like AWS S3 with encryption enabled and restricted access), are the best practice for securing sensitive values in Terraform state files. These backends securely manage state files, including encryption and access control, preventing unauthorized access to sensitive information that may be stored in state files.

Resources
Sensitive Data in State

Question 61 **Incorrect**
Which two steps are required to provision new infrastructure in the Terraform workflow? **(Choose two.)**

Your selection is incorrect
Import
Explanation
❌ The Import step is used to import existing infrastructure into Terraform's state management, not to create new infrastructure.

Correct selection

Apply

Explanation

Apply is a crucial step in the Terraform workflow to provision new infrastructure. The Apply step executes the changes defined in the Terraform configuration files and creates the specified infrastructure resources.

Destroy

Explanation

❌ The Destroy step is used to remove existing infrastructure provisioned by Terraform, not to create new infrastructure.

Validate

Explanation

❌ The Validate step is used to check the syntax and validity of the Terraform configuration files, but it does not actually create any infrastructure resources.

Your selection is correct

Init

Explanation

Init is an essential step in the Terraform workflow to provision new infrastructure. The Init step initializes the Terraform working directory and downloads any necessary plugins and modules required for the configuration.

Overall explanation

Answer: Apply

Answer: Init

Explanation:

To provision new infrastructure using Terraform, the following two steps are essential:

- **Apply**: This command (`terraform apply`) is used to create or update infrastructure based on the configurations defined in the `.tf` files. It executes the plan generated by Terraform and makes the changes in the target environment.
- **Init**: The `terraform init` command initializes the working directory by downloading the necessary provider plugins and setting up the backend configuration. This step must be completed before any other commands, including `apply`.

Incorrect Options:

- **Destroy**: This command (`terraform destroy`) is used to remove existing infrastructure, not to provision new resources.
- **Import**: The `terraform import` command is used to bring existing resources under Terraform management, rather than provisioning new infrastructure.
- **Validate**: While `terraform validate` checks the syntax and validity of the configuration files, it does not provision any infrastructure.

Resources

The Core Terraform Workflow

Question 62 Skipped

Please fill the blank field(s) in the statement with the right words.

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.

Correct answer

-out=FILE

Explanation

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.

Resources

Command: plan

Question 63 Incorrect

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?

Correct answer

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

Explanation

Since the Terraform state file only tracks the new VM you created, executing terraform destroy will specifically target and remove that VM without affecting the other existing VMs in the shared team account. This is the correct way to delete the newly-created VM using Terraform.

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

Explanation

Manually deleting the VM using the cloud provider console and then applying changes with terraform apply is not the recommended approach. Terraform should be used as the primary tool for managing infrastructure to ensure consistency and avoid configuration drift. Manually deleting resources can lead to discrepancies between the Terraform state and the actual infrastructure.

Delete the Terraform state file and execute Terraform apply.

Explanation

Deleting the Terraform state file and then running Terraform apply is not the appropriate method to delete the newly-created VM. This action could lead to inconsistencies in the Terraform configuration and may not effectively remove the specific VM you want to delete.

Your answer is incorrect

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

Overall explanation

Answer: The Terraform state file only contains the one new VM. Execute `terraform destroy`.

Explanation:

In this scenario, the Terraform state file only includes the VM you created with Terraform because it only tracks resources that are part of its configuration. Running `terraform destroy` will remove only the VM defined in your Terraform configuration. Other VMs in the shared account, which were not managed by Terraform, will not be affected.

Incorrect Options:

- The state file does not include all VMs in the account—only those managed by your Terraform configuration.
- Deleting the state file and running `terraform apply` would attempt to recreate resources rather than delete them.
- Manually deleting the VM in the cloud provider console will not update the Terraform state. Terraform will still consider the VM present unless you run `terraform state rm` to remove it from the state.

Question 64 Correct

Which task does `terraform init` **not** perform?

Sources all providers present in the configuration and ensures they are downloaded and available locally

Explanation

Terraform init does indeed source all providers present in the configuration and ensures they are downloaded and available locally. This step is crucial for Terraform to be able to interact with the necessary resources during the deployment process.

Your answer is correct

Validates all required variables are present

Explanation

Terraform init does not validate all required variables are present. This step is typically done during the terraform plan or terraform apply stages, where Terraform checks if all required variables have been provided before proceeding with the deployment.

Connects to the backend

Explanation

Connecting to the backend is an essential part of the terraform init process. The backend configuration specifies where state data is stored and how operations should be executed, making it a key step in setting up the Terraform environment.

Sources any modules and copies the configuration locally.

Explanation

Terraform init does source any modules specified in the configuration and copies the module configuration locally. Modules allow for reusable and shareable configurations, and initializing them is necessary for Terraform to understand and use the module resources.

Overall explanation

Answer: Validates all required variables are present

Explanation:

The `terraform init` command is responsible for initializing a Terraform working directory by performing several tasks, such as downloading provider plugins, connecting to the backend, and sourcing modules. However, it does not validate whether all required variables are present; that task is performed by the `terraform validate` command.

Incorrect Options:

- **Sources all providers present in the configuration and ensures they are downloaded and available locally**: This is a key function of `terraform init`.
- **Connects to the backend**: `terraform init` establishes a connection to the configured backend to store the state.
- **Sources any modules and copies the configuration locally**: `terraform init` also fetches the modules specified in the configuration.

Resources

Terraform Init Command

Question 65Incorrect

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

Using the workspace as a data source

Explanation

Using the workspace as a data source is a capability that is available in both Terraform Enterprise/Cloud workspaces and the Terraform CLI. This feature allows users to reference workspace-specific information and attributes within their Terraform configurations, enabling dynamic and flexible infrastructure management.

Dry runs with terraform plan

Explanation

Dry runs with terraform plan is a functionality that is available in both Terraform Enterprise/Cloud workspaces and the Terraform CLI. This feature allows users to preview the changes that Terraform will make to the infrastructure before actually applying those changes, helping to prevent unintended modifications.

Correct answer

Secure variable storage

Explanation

Secure variable storage is a feature that is only available in Terraform Enterprise or Cloud workspaces, not in the Terraform CLI. This feature allows for the secure storage and management of sensitive variables and secrets within the Terraform environment, providing an added layer of security for infrastructure deployments.

Your answer is incorrect

Support for multiple cloud providers

Explanation

Support for multiple cloud providers is a feature that is available in both Terraform Enterprise/Cloud workspaces and the Terraform CLI. This feature allows users to manage resources across different cloud providers within a single Terraform configuration, enabling multi-cloud infrastructure management.

Overall explanation

Answer: Secure variable storage

Explanation:

Secure variable storage is a feature that is available only in Terraform Enterprise or Terraform Cloud workspaces. This feature allows you to store sensitive variables securely and manage them through the web interface, ensuring that sensitive data is not exposed in your version control system.

Incorrect Options:

- **Support for multiple cloud providers**: This feature is available in both Terraform CLI and Terraform Enterprise/Cloud. Terraform supports multiple cloud providers regardless of the environment.
- **Dry runs with terraform plan**: The `terraform plan` command, which allows you to preview changes without applying them, is available in both Terraform CLI and Terraform Enterprise/Cloud.
- **Using the workspace as a data source**: This capability is also available in both environments, allowing you to reference workspace-specific information in your configurations.

Resources

Terraform Enterprise

Question 66 **Incorrect**

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

Your answer is incorrect

Self-describing infrastructure

Explanation

Self-describing infrastructure is a key principle of infrastructure as code, where infrastructure configurations are written in a way that clearly defines the desired state of the infrastructure. This allows for easy understanding, documentation, and automation of infrastructure changes.

Correct answer

Golden images

Explanation

Golden images are not a key principle of infrastructure as code. Golden images refer to pre-configured virtual machine images that are used as a template for creating new instances. While they can be used in infrastructure provisioning, they are not a fundamental concept in infrastructure as code practices.

Versioned infrastructure

Explanation

Versioned infrastructure is a key principle of infrastructure as code, as it involves managing infrastructure configurations and changes using version control systems like Git. This allows for tracking changes, rolling back to previous versions, and collaborating on infrastructure changes effectively.

Idempotence

Explanation

Idempotence is a key principle of infrastructure as code, emphasizing that the same configuration should result in the same desired state regardless of the number of times it is applied. This ensures consistency and predictability in infrastructure deployments and updates.

Overall explanation

Answer: Golden images

Explanation:

"Golden images" are pre-configured snapshots of operating systems or applications used as a base for deployments. While useful in traditional IT environments, they are not a core principle of Infrastructure as Code (IaC). IaC relies on defining infrastructure in code to enable automated, consistent, and versioned deployments without relying on pre-configured images.

Key principles of Infrastructure as Code include:

- **Versioned infrastructure**: Infrastructure is version-controlled, allowing for rollbacks, tracking changes, and collaboration.
- **Idempotence**: IaC scripts can be executed multiple times without causing unintended changes, ensuring consistency.
- **Self-describing infrastructure**: The code is clear and easy to understand, serving as documentation for the infrastructure.

Question 67Correct

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

servers = variable.num_servers

Explanation

This choice is incorrect because it does not reference the variable correctly. The correct way to access a variable in Terraform is by using the 'var' keyword followed by the variable name.

Your answer is correct

servers = var.num_servers

Explanation

This choice is correct because it correctly references the variable 'num_servers' using the 'var' keyword. In Terraform, variables are accessed using 'var.variable_name'.

servers = var(num_servers)

Explanation

This choice is incorrect because it does not reference the variable correctly. In Terraform, variables are accessed using the 'var' keyword, not 'var()'.

servers = num_servers

Explanation

This choice is incorrect because it does not reference the variable correctly. The variable should be accessed using the 'var' keyword in Terraform.

Overall explanation

Answer: servers = var.num_servers

Explanation:

In Terraform, when you want to pass a variable value into a module, you use the `var.` prefix to access the variable. Therefore, the correct way to pass the value in the variable `num_servers` into a module with the input `servers` is:

- **`servers = var.num_servers`**: This syntax correctly references the variable `num_servers` from the root module and passes its value to the `servers` input of the module.

Incorrect Options:

- **`servers = num_servers`**: This would not work because it tries to reference the variable directly without the `var.` prefix.
- **`servers = variable.num_servers`**: This syntax is incorrect. You should use `var.` instead of `variable.` to access the variable value.
- **`servers = var(num_servers)`**: This is not the correct syntax. The correct way to access a variable is with `var.variable_name` without parentheses.

Resources