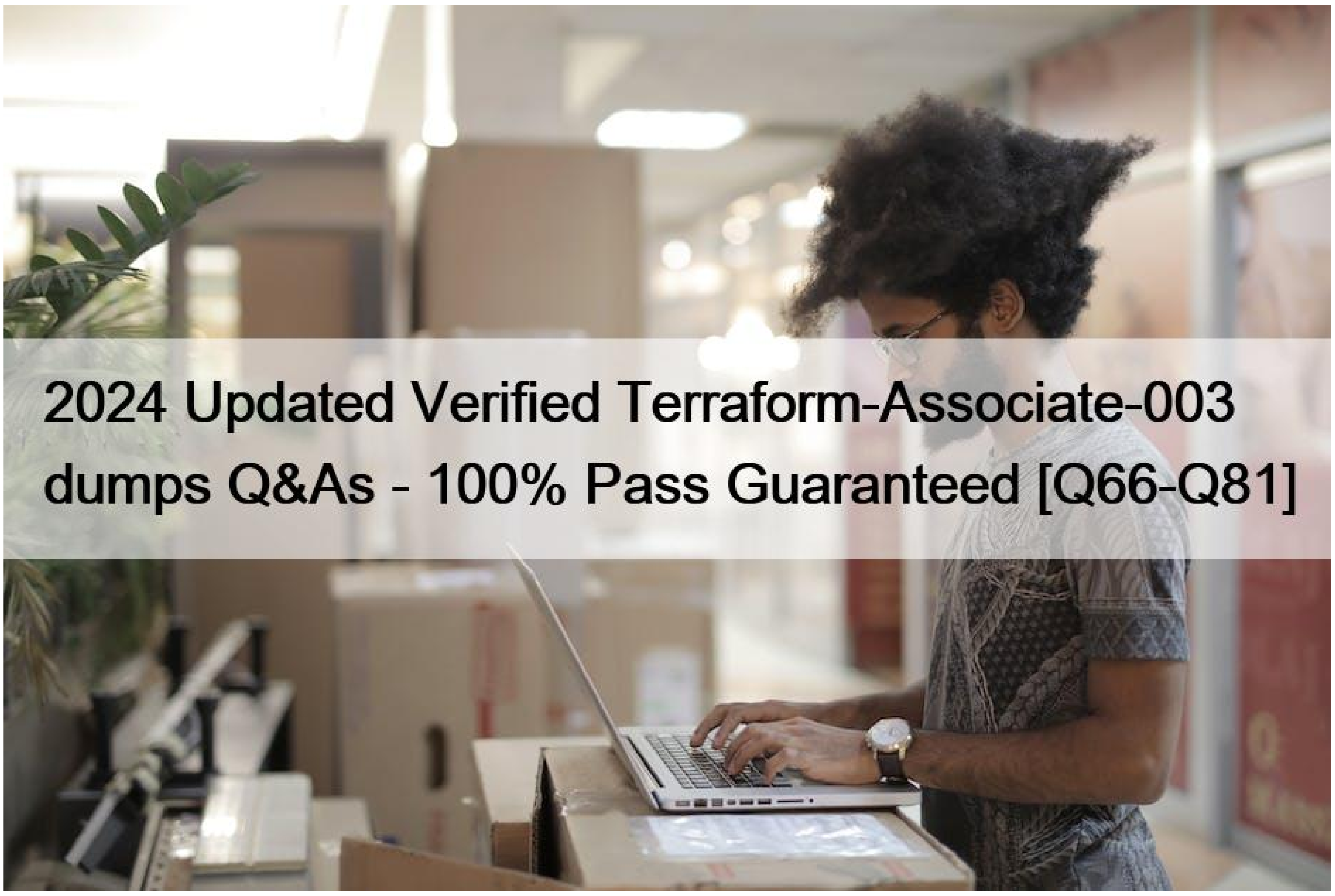
2024 Updated Verified Terraform-Associate-003 dumps Q&As - 100% Pass Guaranteed [Q66-Q81


2024 Updated Verified Terraform-Associate-003 dumps Q&As - 100% Pass Guaranteed [Q66-Q81]

2024 Updated Verified Terraform-Associate-003 dumps Q&As - 100% Pass Guaranteed Provide Valid Dumps To Help You Prepare For HashiCorp Certified: Terraform Associate (003) Exam

HashiCorp Terraform-Associate-003 Exam Syllabus Topics:

TopicDetailsTopic 1- Manage resource lifecycle: The section covers topics such as Initializing a configuration using terraform init and its options and generating an execution plan using terraform plan and its options. It also covers the configuration changes using Terraform Apply and its options.Topic 2- Develop collaborative Terraform workflows: In this section, candidates are tested for their skills related to managing the Terraform binary, providers, and modules using version constraints and setting up remote states. It also covers the utilization of the Terraform workflow in automation.Topic 3- Create, maintain, and use Terraform modules: In this section of the exam, candidates are tested for creating a module, using a module in configuration, and topics such as refactoring an existing configuration into modules.

Q66. Once you configure a new Terraform backend with a terraform code block, which command(s) should you use to migrate the state file?
* terraform destroy, then terraform apply

* terraform init
* terraform push
* terraform apply

Explanation

This command will initialize the new backend and prompt you to migrate the existing state file to the new location4. The other commands are not relevant for this task.

Q67. Which of the following is not a valid source path for specifying a module?
* source &#8211; &#8220;github.com/hashicorp/examplePref-ul.0.8M
* source = &#8220;./module?version=vl.6.0&#8221;
* source &#8211; &#8220;hashicorp/consul/aws&#8221;
* source &#8211; &#8220;./module&#8221;

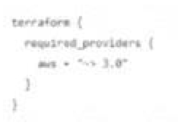
Terraform modules are referenced by specifying a source location. This location can be a URL or a file path. However, specifying query parameters such as ?version=vl.6.0 directly within the source path is not a valid or supported method for specifying a module version in Terraform. Instead, version constraints are specified using the version argument within the module block, not as part of the source string.

Reference = This clarification is based on Terraform&#8217;s official documentation regarding module usage, which outlines the correct methods for specifying module sources and versions.

Q68. Terraform providers are part of the Terraform core binary.
* True
* False

Terraform providers are not part of the Terraform core binary. Providers are distributed separately from Terraform itself and have their own release cadence and version numbers. Providers are plugins that Terraform uses to interact with various APIs, such as cloud providers, SaaS providers, and other services. You can find and install providers from the Terraform Registry, which hosts providers for most major infrastructure platforms. You can also load providers from a local mirror or cache, or develop your own custom providers. To use a provider in your Terraform configuration, you need to declare it in the provider requirements block and optionally configure its settings in the provider block. Reference = : Providers &#8211; Configuration Language | Terraform : Terraform Registry &#8211; Providers Overview | Terraform

Q69. What does this code do?



```
terraform {
  required_providers {
    aws = "~> 3.0"
  }
}
```

* Requires any version of the AWS provider > = 3.0 and <4.0
* Requires any version of the AWS provider >= 3.0
* Requires any version of the AWS provider > = 3.0 major release. like 4.1
* Requires any version of the AWS provider > 3.0

This is what this code does, by using the pessimistic constraint operator (~>), which specifies an acceptable range of versions for a provider or module.

Q70. Only the user that generated a plan may apply it.
* True
* False

Explanation

Any user with permission to apply a plan can apply it, not only the user that generated it. This allows for collaboration and delegation of tasks among team members.

Q71. You have created a main.tf Terraform configuration consisting of an application server, a database and a load balanced. You ran terraform apply and Terraform created all of the resources successfully.

Now you realize that you do not actually need the load balancer, so you run terraform destroy without any flags. What will happen?
* Terraform will prompt you to pick which resource you want to destroy
* Terraform will destroy the application server because it is listed first in the code
* Terraform will prompt you to confirm that you want to destroy all the infrastructure
* Terraform will destroy the main, tf file
* Terraform will immediately destroy all the infrastructure

Explanation

This is what will happen if you run terraform destroy without any flags, as it will attempt to delete all the resources that are associated with your current working directory or workspace. You can use the -target flag to specify a particular resource that you want to destroy.

Q72. 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?
* Run the terraform fmt command during the code linting phase of your CI/CD process Most Voted
* Designate one person in each team to review and format everyone&#8217;s code
* Manually apply two spaces indentation and align equal sign &#8220;=&#8221; characters in every Terraform file (*.tf)
* Write a shell script to transform Terraform files using tools such as AWK, Python, and sed

The terraform fmt command is used to rewrite Terraform configuration files to a canonical format and style.

This command applies a subset of the Terraform language style conventions, along with other minor adjustments for readability. Running this command on your configuration files before committing them to source control can help ensure consistency of style between different Terraform codebases, and can also make diffs easier to read. You can also use the -check and -diff options to check if the files are formatted and display the formatting changes respectively2. Running the terraform fmt command during the code linting phase of your CI/CD process can help automate this process and enforce the formatting standards for your team.
References = [Command: fmt]2

Q73. A Terraform provider is NOT responsible for:
* Exposing resources and data sources based on an APUI
* Managing actions to take based on resources differences
* Understanding API interactions with some service
* Provisioning infrastructure in multiple

Explanation

This is not a responsibility of a Terraform provider, as it does not make sense grammatically or logically. A Terraform provider is responsible for exposing resources and data sources based on an API, managing actions to take based on resource differences, and understanding API interactions with some service.

Q74. How can you trigger a run in a Terraform Cloud workspace that is connected to a Version Control System (VCS) repository?
* Only Terraform Cloud organization owners can set workspace variables on VCS connected workspaces

* Commit a change to the VCS working directory and branch that the Terraform Cloud workspace is connected to
* Only Terraform Cloud organization owners can approve plans in VCS connected workspaces
* Only members of a VCS organization can open a pull request against repositories that are connected to Terraform Cloud workspaces

This will trigger a run in the Terraform Cloud workspace, which will perform a plan and apply operation on the infrastructure defined by the Terraform configuration files in the VCS repository.

Q75. Which of these ate secure options for storing secrets for connecting to a Terraform remote backend? Choose two correct answers.

* A variable file
* Defined in Environment variables
* Inside the backend block within the Terraform configuration
* Defined in a connection configuration outside of Terraform

Environment variables and connection configurations outside of Terraform are secure options for storing secrets for connecting to a Terraform remote backend. Environment variables can be used to set values for input variables that contain secrets, such as backend access keys or tokens. Terraform will read environment variables that start with TF_VAR_ and match the name of an input variable. For example, if you have an input variable called backend_token, you can set its value with the environment variable TF_VAR_backend_token1. Connection configurations outside of Terraform are files or scripts that provide credentials or other information for Terraform to connect to a remote backend. For example, you can use a credentials file for the S3 backend2, or a shell script for the HTTP backend3. These files or scripts are not part of the Terraform configuration and can be stored securely in a separate location. The other options are not secure for storing secrets. A variable file is a file that contains values for input variables. Variable files are usually stored in the same directory as the Terraform configuration or in a version control system. This exposes the secrets to anyone who can access the files or the repository. You should not store secrets in variable files1. Inside the backend block within the Terraform configuration is where you specify the type and settings of the remote backend. The backend block is part of the Terraform configuration and is usually stored in a version control system. This exposes the secrets to anyone who can access the configuration or the repository. You should not store secrets in the backend block4. Reference = [Terraform Input Variables]1, [Backend Type: s3]2, [Backend Type: http]3, [Backend Configuration]4

Q76. You should run terraform fnt to rewrite all Terraform configurations within the current working directory to conform to Terraform-style conventions.

* True
* False

You should run terraform fmt to rewrite all Terraform configurations within the current working directory to conform to Terraform-style conventions. This command applies a subsetof the Terraform language style conventions, along with other minor adjustments for readability. It is recommended to use this command to ensure consistency of style across different Terraform codebases. The command is optional, opinionated, and has no customization options, but it can help you and your team understand the code more quickly and easily. References = : Command: fmt : Using Terraform fmt Command to Format Your Terraform Code

Q77. Which of these statements about Terraform Cloud workspaces is false?

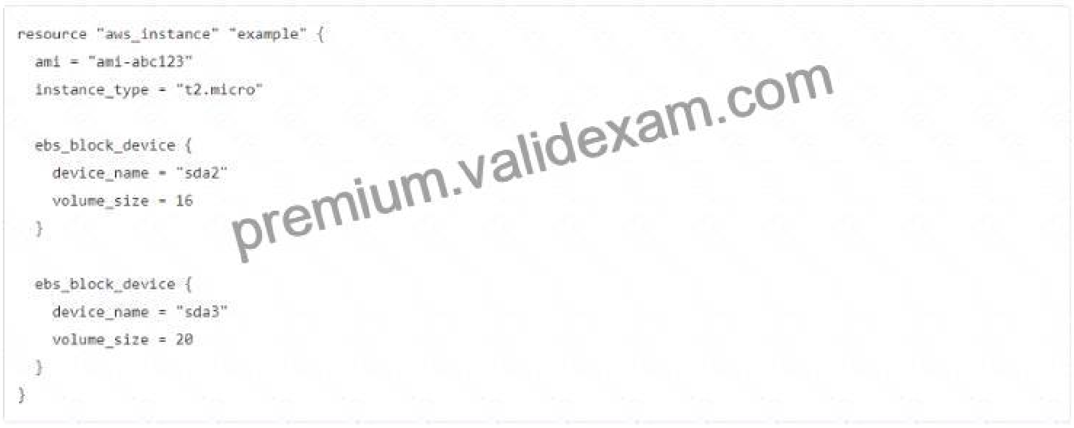
* They have role-based access controls
* You must use the CLI to switch between workspaces
* Plans and applies can be triggered via version control system integrations
* They can securely store cloud credentials

The statement that you must use the CLI to switch between workspaces is false. Terraform Cloud workspaces are different from Terraform CLI workspaces. Terraform Cloud workspaces are required and represent all of the collections of infrastructure in an organization. They are also a major component of role-based access in Terraform Cloud. You can grant individual users and user groups permissions for one or more workspaces that dictate whether they can manage variables, perform runs, etc. You can create, view, and switch between Terraform Cloud workspaces using the Terraform Cloud UI, the Workspaces API, or the Terraform Enterprise Provider5. Terraform CLI workspaces are optional and allow you to create multiple distinct instances of a single configuration within one working directory. They are useful for creating disposable environments for testing or experimenting without affecting your main or production environment. You can create, view, and switch between Terraform CLI workspaces using the terraform workspace command6. The other statements about Terraform Cloud workspaces are true. They have role-based access controls that allow you to assign permissions to users and teams based on their roles and responsibilities. You can create and manage roles using the Teams API or the Terraform Enterprise Provider7. Plans and applies can be triggered via version control system integrations that allow you to link your Terraform Cloud workspaces to your VCS repositories. You can configure VCS settings, webhooks, and branch tracking to automate your Terraform Cloud workflow8. They can securely store cloud credentials as sensitive variables that are encrypted at rest and only decrypted when needed. You can manage variables using the Terraform Cloud UI, the Variables API, or the Terraform Enterprise Provider9. Reference = [Workspaces]5, [Terraform CLI Workspaces]6, [Teams and Organizations]7, [VCS Integration]8, [Variables]9

Q78. Which of the following statements about Terraform modules is not true?
* Modules can call other modules
* A module is a container for one or more resources
* Modules must be publicly accessible
* You can call the same module multiple times
Explanation

This is not true, as modules can be either public or private, depending on your needs and preferences. You can use the Terraform Registry to publish and consume public modules, or use Terraform Cloud or Terraform Enterprise to host and manage private modules.

Q79. How would you reference the volume IDs associated with the ebs_block_device blocks in this configuration?



```
resource "aws_instance" "example" {
  ami = "ami-abc123"
  instance_type = "t2.micro"

  ebs_block_device {
    device_name = "sda2"
    volume_size = 16
  }

  ebs_block_device {
    device_name = "sda3"
    volume_size = 20
  }
}
```

* aws_instance.example.ebs_block_device[sda2,sda3).volume_id
* aws_lnstance.example.ebs_block_device.[*].volume_id
* aws_lnstance.example.ebs_block_device.volume_ids
* aws_instance.example-ebs_block_device.*.volume_id
This is the correct way to reference the volume IDs associated with the ebs_block_device blocks in this configuration, using the splat expression syntax. The other options are either invalid or incomplete.

Q80. A developer accidentally launched a VM (virtual machine) outside of the Terraform workflow and ended up with two servers with the same name. They don&#8217;t know which VM Terraform manages but do have a list of all active VM IDs.

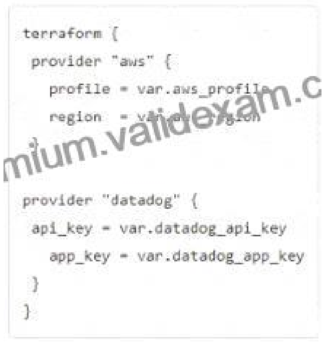
Which of the following methods could you use to discover which instance Terraform manages?
* Run terraform state list to find the names of all VMs, then run terraform state show for each of them to find which VM ID Terraform manages

* Update the code to include outputs for the ID of all VMs, then run terraform plan to view the outputs
* Run terraform taint/code on all the VMs to recreate them
* Use terraform refresh/code to find out which IDs are already part of state

Explanation

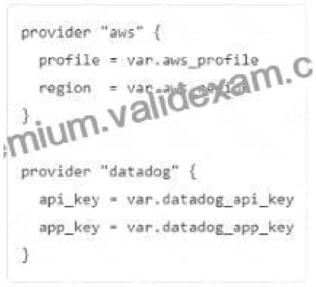
The terraform state list command lists all resources that are managed by Terraform in the current state file1. The terraform state show command shows the attributes of a single resource in the state file2. By using these two commands, you can compare the VM IDs in your list with the ones in the state file and identify which one is managed by Terraform.

Q81. You ate creating a Terraform configuration which needs to make use of multiple providers, one for AWS and one for Datadog. Which of the following provider blocks would allow you to do this?

* 

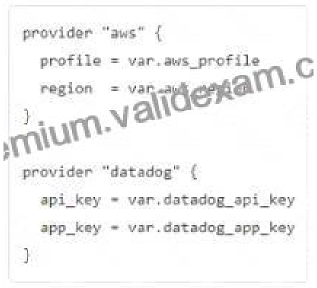
```
terraform {
 provider "aws" {
   profile = var.aws_profile
   region  = var.aws_region
 }

provider "datadog" {
 api_key = var.datadog_api_key
   app_key = var.datadog_app_key
 }
}
```

* 

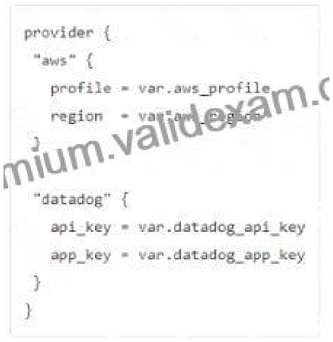
```
provider "aws" {
  profile = var.aws_profile
  region  = var.aws_region
}

provider "datadog" {
  api_key = var.datadog_api_key
  app_key = var.datadog_app_key
}
```

* 

```
provider "aws" {
  profile = var.aws_profile
  region  = var.aws_region
}

provider "datadog" {
  api_key = var.datadog_api_key
  app_key = var.datadog_app_key
}
```

* 

```
provider {
 "aws" {
   profile = var.aws_profile
   region  = var.aws_region
 }

 "datadog" {
   api_key = var.datadog_api_key
   app_key = var.datadog_app_key
 }
}
```

Option C is the correct way to configure multiple providers in a Terraform configuration. Each provider block must have a name attribute that specifies which provider it configures2. The other options are either missing the name attribute or using an invalid syntax.