Demo

HashiCorp Terraform-Associate Exam Dumps

HashiCorp Terraform-Associate Dumps

CLOUD EXAM DUMPS

Question 1: Terraform Cloud is more powerful when you integrate it with your version control system (VCS) provider. Select all the supported VCS providers from the answers below. (select four)

A. Bitbucket Cloud	**(Correct)**
B. GitHub.com	**(Correct)**
C. CVS Version Control	
D. Azure DevOps Server	**(Correct)**
E. GitHub Enterprise	**(Correct)**

Explanation
Terraform Cloud supports the following VCS providers as of February 2023:

- GitHub
- GitHub.com (OAuth)
- GitHub Enterprise
- GitLab.com
- GitLab EE and CE
- Bitbucket Cloud
- Bitbucket Server
- Azure DevOps Server
- Azure DevOps Services

https://developer.hashicorp.com/terraform/cloud-docs/vcs#supported-vcs-providers

Question 2: From the code below, identify the ***implicit*** dependency:

```
resource "aws_eip" "public_ip" {
   vpc = true
   instance = aws_instance.web_server.id
}

resource "aws_instance" "web_server" {
 ami           = "ami-2757f631"
 instance_type = "t2.micro"
 depends_on = [aws_s3_bucket.company_data]
```

```
}
```

A. The EC2 instance labeled `web_server` (Correct)

B. The S3 bucket labeled `company_data`

C. The `AMI` used for the EC2 instance

D. The EIP with an id of `ami-2757f631`

Explanation

Implicit dependencies are not explicitly declared in the configuration but are automatically detected by Terraform based on the relationships between resources. Implicit dependencies allow Terraform to automatically determine the correct order in which resources should be created, updated, or deleted, ensuring that resources are created in the right order, and dependencies are satisfied.

For example, if you have a resource that depends on another resource, Terraform will automatically detect this relationship and create the dependent resource after the resource it depends on has been created. This allows Terraform to manage complex infrastructure deployments in an efficient and predictable way.

The EC2 instance labeled web_server is the implicit dependency as the aws_eip cannot be created until the aws_instance labeled web_server has been provisioned and the id is available.

Note that aws_s3_bucket.company_data is an explicit dependency for the aws_instance.web_server

https://learn.hashicorp.com/tutorials/terraform/dependencies

Question 3:

Which Terraform command will check and report errors within modules, attribute names, and value types to ensure they are syntactically valid and internally consistent?

A. `terraform validate` (Correct)

B. `terraform format`

C. `terraform show`

D. `terraform fmt`

Explanation

The `terraform validate` command is used to check and report errors within modules, attribute names, and value types to ensure they are syntactically valid and internally consistent. This command performs basic validation of the Terraform configuration files in the current directory, checking for issues such as missing required attributes, invalid attribute values, and incorrect structure of the Terraform code.

For example, if you run **`terraform validate`** and there are syntax errors in your Terraform code, Terraform will display an error message indicating the line number and description of the issue. If no errors are found, the command will return with no output.

It's recommended to run **`terraform validate`** before running **`terraform apply`**, to ensure that your Terraform code is valid and will not produce unexpected results.

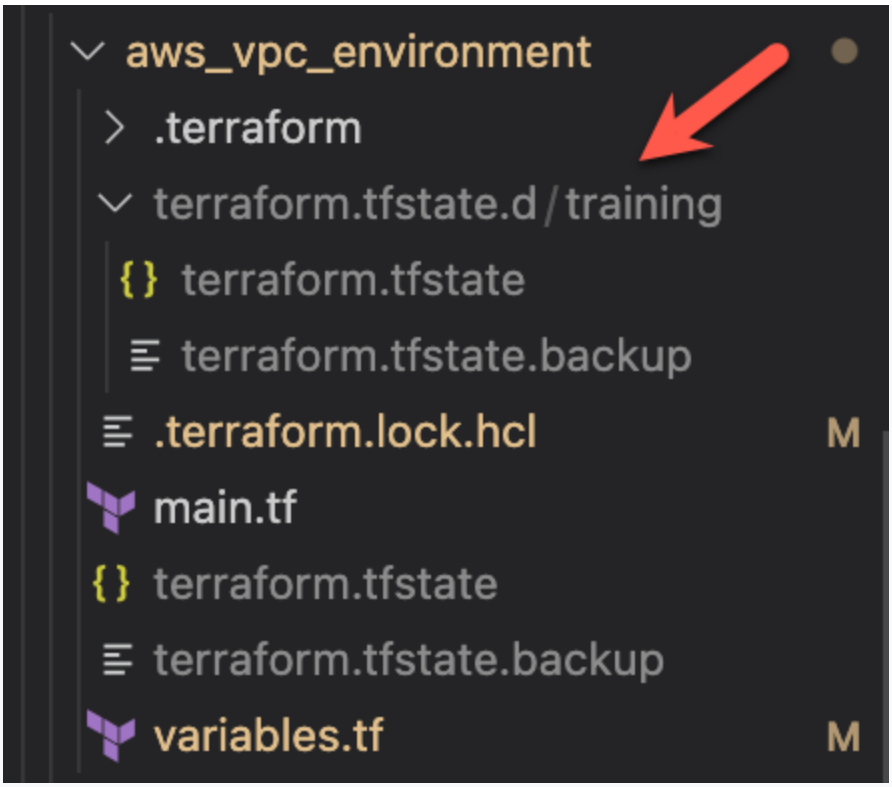
https://developer.hashicorp.com/terraform/cli/commands/validate

Question 4:
Where does Terraform Open Source (OSS) store the *local* state for workspaces?

- **A. a file called `terraform.tfstate.backup`**
- **B. directory called `terraform.tfstate.d/<workspace name>` (Correct)**
- **C. a file called `terraform.tfstate`**
- **D. directory called `terraform.workspaces.tfstate`**

Explanation
Terraform Open Source (OSS) stores the local state for workspaces in a file on disk. For local state, Terraform stores the workspace states in a directory called `terraform.tfstate.d/<workspace_name>`. Here's a screenshot of a Terraform run that was created using a workspace called `training`. You can see that Terraform created the `terraform.tfstate.d` directory, and then a directory with the namespace name underneath it.

Under each directory, you'll find the state file, which is name `terraform.tfstate`



https://developer.hashicorp.com/terraform/cli/workspaces#workspace-internals

Question 5:
You are adding a new variable to your configuration. Which of the following is **NOT** a valid variable type in Terraform?

A. `string`
B. `number`
C. `bool`
D. `float` **(Correct)**
E. `map`

Explanation

The Terraform language uses the following types for its values: `string`, `number`, `bool`, `list` (or tuple), `map` (or object). There are no other supported variable types in Terraform, therefore, `float` is incorrect in this question.

Don't forget that variable types are included in a variable block, but they are NOT required since Terraform interprets the type from a default value or value provided by other means (ENV, CLI flag, etc)

```
variable "practice-exam" {
 description = "bryan's terraform associate practice exams"
 type        = string
 default     = "highly-rated"
}
```

https://developer.hashicorp.com/terraform/language/expressions/types

Question 6:
Harry has deployed resources on Azure for his organization using Terraform. However, he has discovered that his co-workers Ron and Ginny have manually created a few resources using the Azure console. Since it's company policy to manage production workloads using IaC, how can Harry start managing these resources in Terraform without negatively impacting the availability of the deployed resources?

A. use `terraform import` to import the existing resources under Terraform management (Correct)
B. rewrite the Terraform configuration file to deploy new resources, run a `terraform apply`, and migrate users to the newly deployed resources. Manually delete the other resources created by Ron and Ginny.
C. run a `terraform get` to retrieve other resources that are not under Terraform management
D. resources created outside of Terraform cannot be managed by Terraform

Explanation
To manage the resources created manually by Ron and Ginny in Terraform without negatively impacting the availability of the deployed resources, Harry can follow the steps below:

1. Import the existing resources: Harry can use the `terraform import` command to import the existing resources into Terraform. The `terraform import` command allows you to import existing infrastructure into Terraform, creating a Terraform state file for the resources.

2. Modify the Terraform configuration: After importing the resources, Harry can modify the Terraform configuration to reflect the desired state of the resources. This will allow him to manage the resources using Terraform just like any other Terraform-managed resource

3. Test the changes: Before applying the changes, Harry can use the **`terraform plan`** command to preview the changes that will be made to the resources. This will allow him to verify that the changes will not negatively impact the availability of the resources.

4. Apply the changes: If the changes are correct, Harry can use the **`terraform apply`** command to apply the changes to the resources.

By following these steps, Harry can start managing the manually created resources in Terraform while ensuring that the availability of the deployed resources is not impacted.

The `terraform import` command is used to import existing resources into Terraform. This allows you to take resources that you've created by some other means and bring them under Terraform management.

*Note that terraform import **DOES NOT** generate configuration, it only modifies state. You'll still need to write a configuration block for the resource for which it will be mapped using the `terraform import` command.*

https://developer.hashicorp.com/terraform/cli/commands/import

Question 7:
You are developing a new Terraform module to demonstrate features of the most popular HashiCorp products. You need to spin up an AWS instance for each tool, so you create the resource block as shown below using the `for_each` meta-argument.

```
resource "aws_instance" "bryan-demo" {
 # ...
 for_each = {
   "terraform": "infrastructure",
   "vault":     "security",
   "consul":    "connectivity",
   "nomad":     "scheduler",
 }
}
```

After the deployment, you view the state using the `terraform state list` command. What resource address would be displayed for the instance related to `vault`?

A. aws_instance.bryan-demo["2"]
B. aws_instance.bryan-demo["vault"] **(Correct)**
C. aws_instance.bryan-demo.vault
D. aws_instance.bryan-demo[1]

Explanation

In Terraform, when you use the **`for_each`** argument in a resource block, Terraform generates multiple instances of that resource, each with a unique address. The address of each instance is determined by the keys of the **`for_each`** map, and it is used to identify and manage each instance of the resource.

For example, consider the following resource block in the question:

```
resource "aws_instance" "bryan-demo" {
 # ...
 for_each = {
   "terraform": "infrastructure",
   "vault":     "security",
   "consul":    "connectivity",
   "nomad":     "scheduler",
 }
}
```

In this example, Terraform will create four instances of the **`aws_instance`** resource, one for each key in the **`for_each`** map. The addresses of these instances will be **`aws_instance.bryan-demo["terraform"]`** , **`aws_instance.bryan-demo["vault"]`**,**`aws_instance.bryan-demo["consul"]`**, and **`aws_instance.bryan-demo["nomad"]`**.

When you reference the properties of these instances in your Terraform code, you can use the address and property reference syntax to access the properties of each instance. For example, you can access the ID of the first instance using **`aws_instance.bryan-demo["vault"].id`**.

Using the **`for_each`** argument in a resource block is a powerful way to manage multiple instances of a resource, and it provides a convenient way to reuse the same resource configuration for multiple instances with different properties.

https://developer.hashicorp.com/terraform/cli/v1.1.x/state/resource-addressing

Question 8:
True or False? Terraform Enterprise offers the ability to use Terraform to deploy infrastructure in your local on-premises datacenter as well as a public cloud platform, such as AWS, Azure, or GCP.

A. True (Correct)

B. False

Explanation

Terraform Enterprise is often installed on-premises or in a public cloud and provides the ability to deploy resources using any provider/plugin that is supported by Terraform. TFE gives you many features that are not available with Terraform OSS. Many of these features, however, are available in Terraform Cloud, though.

https://developer.hashicorp.com/terraform/cloud-docs/overview

Question 9:

Sara has her entire application automated using Terraform, but she needs to start automating more infrastructure components, such as creating a new subnet, DNS record, and load balancer. Sara wants to create these new resources within modules to easily reuse the code later. However, Sara is having problems getting the `subnet_id` from the *subnet* module to pass to the *load balancer* module.

modules/subnet.tf:

```
resource "aws_subnet" "bryan" {
 vpc_id     = aws_vpc.krausen.id
 cidr_block = "10.0.1.0/24"

 tags = {
   Name = "Krausen Subnet"
 }
}
```

What could fix this problem?

A. add an `output` block that references the *subnet* module and retrieves the value using `module.subnet.subnet_id` in the load balancer module (Correct)

B. publish the module to a Terraform registry first

C. move the *subnet* and *load balancer* resource into the main configuration file so they can easily be referenced

D. references to resources that are created within a module cannot be used within other modules

Explanation

Modules also have output values, which are defined within the module with the `output` keyword. You can access them by referring to `module.<MODULE NAME>.<OUTPUT NAME>`. Like input variables, module outputs are listed under the `outputs` tab in the Terraform registry.

Module outputs are usually either passed to other parts of your configuration, or defined as outputs in your root module.

https://learn.hashicorp.com/tutorials/terraform/module-use#define-root-output-values

Question 10:
What feature of Terraform Cloud allows you to publish and maintain a set of custom modules which can be used within your organization?

A. custom VCS integration
B. remote runs
C. private module registry (Correct)
D. Terraform registry

Explanation
You can use modules from a private registry, like the one provided by Terraform Cloud. Private registry modules have source strings of the form `<HOSTNAME>/<NAMESPACE>/<NAME>/<PROVIDER>`. This is the same format as the public registry, but with an added hostname prefix.

https://www.datocms-assets.com/2885/1602500234-terraform-full-feature-pricing-tablev2-1.pdf

Question 11:
Which are some of the benefits of using Infrastructure as Code in an organization? (select three)

A. IaC allows you to commit your configurations to version control to safely collaborate on infrastructure
(Correct)
B. IaC code can be used to manage infrastructure on multiple cloud platforms
(Correct)
C. IaC is written as an imperative approach, where specific commands need to be executed in the correct order
D. IaC uses a human-readable configuration language to help you write infrastructure code quickly
(Correct)

Explanation

Infrastructure as Code has many benefits. For starters, IaC allows you to create a blueprint of your data center as code that can be ***versioned***, ***shared***, and ***reused***. Because IaC is code, it can (and should) be stored and managed in a **code repository**, such as GitHub, GitLab, or Bitbucket. Changes can be proposed or submitted via Pull Requests (PRs), which can help ensure a proper workflow, enable an approval process, and follow a typical development lifecycle.

One of the primary reasons that Terraform (or other IaC tools) are becoming more popular is because they are mostly ***platform agnostic***. You can use Terraform to provision and manage resources on various platforms, SaaS products, and even local infrastructure.

IaC is generally ***easy to read*** (and develop). Terraform is written in HashiCorp Configuration Language (HCL), while others may use YAML or solution-specific languages (like Microsoft ARM). But generally, IaC code is easy to read and understand

Incorrect Answer:

IaC is written using a ***declarative*** approach (***not imperative***), which allows users to simply focus on what the eventual target configuration should be, and the tool manages the process of how that happens. This often speeds things up because resources can be created/managed in parallel when there aren't any implicit or explicit dependencies.

https://developer.hashicorp.com/terraform/tutorials/aws-get-started/infrastructure-as-code

https://www.terraform.io/use-cases/infrastructure-as-code

Question 12:
You are using Terraform to deploy some cloud resources and have developed the following code. However, you receive an error when trying to provision the resource. Which of the following answer fixes the syntax of the Terraform code?

```
resource "aws_security_group" "vault_elb" {
 name        = "${var.name_prefix}-vault-elb"
 description = Vault ELB
 vpc_id      = var.vpc_id
}
```

A. `resource "aws_security_group" "vault_elb" {`
`name = "${var.name_prefix}-vault-elb"`

```
  description = var_Vault ELB
  vpc_id      = var.vpc_id
}

B. resource "aws_security_group" "vault_elb" {
  name        = "${var.name_prefix}-vault-elb"
  description = "${Vault ELB}"
  vpc_id      = var.vpc_id
}
C. resource "aws_security_group" "vault_elb" {
  name        = "${var.name_prefix}-vault-elb"
  description = "Vault ELB"
  vpc_id      = var.vpc_id
}
```

(Correct)

```
D. resource "aws_security_group" "vault_elb" {
  name        = "${var.name_prefix}-vault-elb"
  description = [Vault ELB]
  vpc_id      = var.vpc_id
}
```

Explanation

When assigning a value to an argument in Terraform, there are a few requirements that must be met:

1. Data type: The value must be of the correct data type for the argument. Terraform supports several data types, including **strings**, numbers, booleans, lists, and maps.
2. Value constraints: Some arguments may have specific value constraints that must be met. For example, an argument may only accept values within a certain range or values from a specific set of values.

When assigning a value to an argument expecting a string, it must be enclosed in quotes ("...") unless it is being generated programmatically.

https://developer.hashicorp.com/terraform/language/syntax/configuration#arguments-and-blocks

Question 13:
When using constraint expressions to signify a version of a provider, which of the following are valid provider versions that satisfy the expression found in the following code snippet: (select two)

```
terraform {
 required_providers {
   aws = "~> 1.2.0"
 }
}
```

A. Terraform 1.2.3 (Correct)
B.Terraform 1.3.1
C. Terraform 1.3.0
D. Terraform 1.2.9 (Correct)

Explanation
In a `required_version` parameter in Terraform, the tilde (`~`) symbol followed by the greater than symbol (`>`) specifies a "compatible with" version constraint.

For example, if your Terraform configuration specifies `required_version = "~> 1.12.0"`, Terraform will accept any version of Terraform 1.12 that is greater than or equal to version 1.12.0 and less than 1.13.0. In other words, Terraform will accept any version of Terraform 1.12 that is considered compatible with version 1.12.0.

This type of version constraint is useful when your Terraform configuration uses features that are available in a specific version of Terraform, but you also want to allow for later versions of Terraform that are compatible with that version. This allows you to specify a minimum required version of Terraform, while also allowing for later versions that are compatible with your configuration.

Note that version constraints specified using the tilde and greater than symbols are specific to Terraform, and they are not a standard part of the Semantic Versioning specification.

https://developer.hashicorp.com/terraform/language/modules/syntax#version

https://developer.hashicorp.com/terraform/language/expressions/version-constraints#version-constraint-syntax

Question 14:
What do the declarations, such as `name`, `cidr`, and `azs`, in the following Terraform code represent and what purpose do they serve?

```
module "vpc" {
 source  = "terraform-aws-modules/vpc/aws"
 version = "2.21.0"

 name = var.vpc_name
 cidr = var.vpc_cidr

 azs             = var.vpc_azs
 private_subnets = var.vpc_private_subnets
 public_subnets  = var.vpc_public_subnets

 enable_nat_gateway = var.vpc_enable_nat_gateway

 tags = var.vpc_tags
}
```

A. these are the `outputs` that the child module will return
B. these are `variables` that are passed into the child module likely used for resource creation (Correct)
C. the `value` of these variables will be obtained from values created within the child module
D. these are where the `variable declarations` are created so Terraform is aware of these variables within the calling module

Explanation
To pass values to a Terraform module when calling the module in your code, you use input variables. Input variables are a way to pass values into a Terraform module from the calling code. They allow the module to be flexible and reusable, as the same module can be used with different input values in different contexts.

In this example, the values for the `name`, `cidr`, and `azs` inputs are passed to the module as values of variables. The variables are defined in the calling code in the calling module using the `variable` block.

To pass the values to the module, you can specify them in a number of ways, such as:

- Using command-line flags when running Terraform
- Storing the values in a Terraform `.tfvars` file and passing that file to Terraform when running it

- Using environment variables

For more information on Terraform modules and input variables, I recommend checking out the Terraform documentation: **https://www.terraform.io/docs/modules/index.html**

https://learn.hashicorp.com/tutorials/terraform/module-use#set-values-for-module-input-variables

Question 15:
Provider dependencies are created in several different ways. Select the valid provider dependencies from the following list: (select three)

A. Explicit use of a provider block in configuration, optionally including a version constraint. **(Correct)**
B. Existence of any resource instance belonging to a particular provider in the current *state*. **(Correct)**
C. Existence of any provider plugins found locally in the working directory.
D. Use of any resource belonging to a particular provider in a resource or data block in the configuration. **(Correct)**

Explanation
The existence of a provider plugin found locally in the working directory does not itself create a provider dependency. The plugin can exist without any reference to it in the Terraform configuration.

https://developer.hashicorp.com/terraform/language/providers/requirements

https://developer.hashicorp.com/terraform/cli/commands/providers

Question 16:
A "backend" in Terraform determines how state is loaded and how an operation such as `apply` is executed. Which of the following is **not** a supported backend type?

A. github **(Correct)**
B. s3
C. local
D. consul

Explanation
GitHub is not a supported backend type.

The Terraform backend is responsible for storing the state of your Terraform infrastructure and ensuring that state is consistent across all team members. Terraform state is used to store information about the resources that Terraform has created, and is used by Terraform to determine what actions are necessary when you run Terraform commands like `apply` or `plan`.

Terraform provides several backend options, including:

- `local` backend: The default backend, which stores Terraform state on the local filesystem. This backend is suitable for small, single-user deployments, but can become a bottleneck as the size of your infrastructure grows or as multiple users start managing the infrastructure.

- `remote` backend: This backend stores Terraform state in a remote location, such as an S3 bucket, a Consul server, or a Terraform Enterprise instance. The remote backend allows multiple users to share the same state and reduces the risk of state corruption due to disk failures or other issues.

- `consul` backend: This backend stores Terraform state in a Consul cluster. Consul provides a highly available and durable storage solution for Terraform state, and also provides features like locking and versioning that are important for collaboration.

- `s3` backend: This backend stores Terraform state in an S3 bucket. S3 provides a highly available and durable storage solution for Terraform state, and is a popular option for storing Terraform state for large infrastructure deployments.

When choosing a backend, you should consider the needs of your infrastructure, including the size of your deployment, the number of users who will be managing the infrastructure, and the level of collaboration that will be required. It's also important to consider the cost and performance characteristics of each backend, as some backends may be more expensive or may require more setup and maintenance than others.

https://developer.hashicorp.com/terraform/language/settings/backends/configuration

Question 17:
Henry has been working on automating his Azure infrastructure for a new application using Terraform. His application runs successfully, but he has added a new resource to create a DNS record using the new Infoblox provider. He has added the new resource but gets an error when he runs a `terraform plan`.

What should Henry do first before running a `plan` and `apply`?

A. since a new provider has been introduced, `terraform init` needs to be run to download the Infoblox plugin (Correct)

B. Henry should run a `terraform plan -refresh=true` to update the state for the new DNS resource

C. the Azure plugin doesn't support Infoblox directly, so Henry needs to put the DNS resource in another configuration file

D. you can't mix resources from different providers within the same configuration file, so Henry should create a module for the DNS resource and reference it from the main configuration

Explanation

In this scenario, Henry has introduced a new provider. Therefore, Terraform needs to download the plugin to support the new resource he has added. Running **`terraform init`** will download the Infoblox plugin. Once that is complete, a `plan` and `apply` can be executed as needed.

You would need to rerun **`terraform init`** after modifying your code for the following reasons:

1. **Adding a new provider**: If you've added a new provider to your code, you'll need to run **`terraform init`** to download the provider's binary and configure it.

2. Updating the provider configuration: If you've updated the configuration of an existing provider, you'll need to run **`terraform init`** to apply the changes.

3. Updating the version of a provider: If you've updated the version of a provider, you'll need to run **`terraform init`** to download the updated version of the provider's binary.

4. Adding or removing a module: If you've added or removed a module from your code, you'll need to run **`terraform init`** to download the required modules and dependencies.

In short, **`terraform init`** is used to initialize a Terraform working directory, and you'll need to rerun it whenever you make changes to your code that affect the providers, modules, or versions you're using.

https://developer.hashicorp.com/terraform/cli/commands/init

Question 18: In the example below, the `depends_on` argument creates what type of dependency?

```
resource "aws_instance" "example" {
 ami           = "ami-2757f631"
 instance_type = "t2.micro"
 depends_on = [aws_s3_bucket.company_data]
}
```

A. implicit dependency
B. explicit dependency (Correct)
C. internal dependency
D. non-dependency resource

Explanation
Explicit dependencies in Terraform are dependencies that are explicitly declared in the Terraform configuration. These dependencies are used to control the order in which Terraform creates, updates, and destroys resources.

In Terraform, you can declare explicit dependencies using the **`depends_on`** argument in a resource block. The **`depends_on`** argument takes a list of resource names and specifies that the resource block in which it is declared depends on those resources.

For example, consider a scenario where you have a virtual machine (VM) that depends on a virtual network (VNET) and a subnet. You can declare these dependencies using the **`depends_on`** argument as follows:

```
resource "azurerm_virtual_network" "vnet" {
 name                = "example-vnet"
 address_space       = ["10.0.0.0/16"]
}

resource "azurerm_subnet" "subnet" {
 name                 = "example-subnet"
 virtual_network_name = azurerm_virtual_network.vnet.name
```

```
 address_prefix       = "10.0.1.0/24"
}

resource "azurerm_network_interface" "nic" {
 name                = "example-nic"
 location            =
azurerm_virtual_network.vnet.location
 subnet_id           = azurerm_subnet.subnet.id
 depends_on = [
   azurerm_subnet.subnet,
   azurerm_virtual_network.vnet
 ]
}
```

In this example, the `azurerm_network_interface` resource depends on both the `azurerm_subnet` and the `azurerm_virtual_network` resources, so Terraform will create those resources first, and then create the `azurerm_network_interface` resource.

By declaring explicit dependencies, you can ensure that Terraform creates resources in the correct order, so that dependent resources are available before other resources that depend on them. This helps prevent errors or unexpected behavior when creating or modifying infrastructure, and makes it easier to manage and understand the relationship between resources.

Overall, the use of explicit dependencies is a critical aspect of Terraform, as it helps ensure that resources are created and managed in the correct order and makes it easier to manage and understand the relationship between resources.

https://learn.hashicorp.com/tutorials/terraform/dependencies

Question 19:
True or False? Rather than use a state file, Terraform can inspect cloud resources on every run to validate that the real-world resources match the desired state.

A. False (Correct)
B. True

Explanation
State is a necessary requirement for Terraform to function. And in the scenarios where Terraform may be able to get away without state, doing so would require shifting

massive amounts of complexity from one place (state) to another place (the replacement concept).

To support mapping configuration to resources in the real world, Terraform uses its own state structure. Terraform can guarantee one-to-one mapping when it creates objects and records their identities in the state. Terraform state also serves as a performance improvement - rather than having to scan every single resource to determine the current state of each resource.

https://developer.hashicorp.com/terraform/language/state/purpose

Question 20:
True or False? When using the Terraform provider for Vault, the tight integration between these HashiCorp tools provides the ability to mask secrets in the state file.

A. True
B. False (Correct)

Explanation
False. By default, Terraform does not provide the ability to mask secrets in the Terraform plan and state files regardless of what provider you are using. While Terraform and Vault are both developed by HashiCorp and have a tight integration, masking secrets in Terraform plans and state files requires additional steps to securely manage sensitive information.

One common approach is to use environment variables or Terraform input variables to store sensitive information, and then use Terraform's data sources to retrieve the information from the environment or input variables, rather than hardcoding the information into the Terraform configuration. This helps to ensure that sensitive information is not stored in plain text in the Terraform plan or state files.

https://learn.hashicorp.com/tutorials/terraform/secrets-vault

Question 21:
True or False? Each Terraform workspace uses its own state file to manage the infrastructure associated with that particular workspace.

A. False
B. True (Correct)

Explanation
True. Each Terraform workspace uses its own state file to manage the infrastructure associated with that workspace. This allows Terraform to manage multiple sets of infrastructure independently and avoid conflicts. Each Terraform workspace has its own Terraform state file that keeps track of the resources and their attributes, so changes made in one workspace will not affect the infrastructure managed by other workspaces.

In fact, having different state files provides the benefits of workspaces, where you can separate the management of infrastructure resources so you can make changes to specific resources without impacting resources in others....

https://developer.hashicorp.com/terraform/language/state/workspaces#workspace-internals

Question 22:
When writing Terraform code, how many spaces between each nesting level does HashiCorp recommends that you use?

A. 1
B. 5
C. 2 (Correct)
D. 4

Explanation
HashiCorp, the creator of Terraform, recommends using **two spaces** for indentation when writing Terraform code. This is a convention that helps to improve readability and consistency across Terraform configurations.

For example, when defining a resource in Terraform, you would use two spaces to indent each level of the resource definition, as in the following example:

```
resource "aws_instance" "example" {
 ami           = "ami-0c55b159cbfafe1f0"
 instance_type = "t2.micro"

 tags = {
   Name = "example-instance"
 }
}
```

While this is the recommended convention, ***it is not a strict requirement*** and Terraform will still function correctly even if you use a different number of spaces or a different type of indentation. However, using two spaces for indentation is a widely adopted convention in the Terraform community and is recommended by HashiCorp to improve the readability and maintainability of your Terraform configurations.

Check this link for more information

Question 23:
A user creates three workspaces from the command line: `prod`, `dev`, and `test`. Which of the following commands will the user run to switch to the `dev` workspace?

A. `terraform workspace -switch dev`
B. `terraform workspace select dev` **(Correct)**
C. `terraform workspace switch dev`
D. `terraform workspace dev`

Explanation
The command used to switch to the dev workspace in Terraform is **`terraform workspace select dev`**.

Terraform workspaces allow you to manage multiple sets of infrastructure resources that share the same configuration. To switch to a specific workspace in Terraform, you use the **`terraform workspace select`** command followed by the name of the workspace you want to switch to. In this case, the name of the workspace is "dev".

After running this command, Terraform will switch to the **`dev`** workspace, and all subsequent Terraform commands will apply to the resources in that workspace. If the **`dev`** workspace does not yet exist, Terraform will **NOT** create it for you.

Here's an example of using the **`terraform workspace select`** command to switch to the dev workspace:

```
$ terraform workspace select dev
Switched to workspace "dev".
```

https://developer.hashicorp.com/terraform/cli/commands/workspace/select

Question 24:
True or False? By default, the `terraform destroy` command will prompt the user for confirmation before proceeding.

A. True (Correct)
B. False

Explanation
True. By default, Terraform will prompt for confirmation before proceeding with the **`terraform destroy`** command. This prompt allows you to verify that you really want to destroy the infrastructure that Terraform is managing before it actually does so.

Terraform destroy will always prompt for confirmation before executing unless passed the **`-auto-approve`** flag.

```
$ terraform destroy
Do you really want to destroy all resources?
 Terraform will destroy all your managed infrastructure, as
shown above.
 There is no undo. Only 'yes' will be accepted to confirm.

 Enter a value: yes
```

https://developer.hashicorp.com/terraform/tutorials/aws-get-started/aws-destroy

Question 25:
Which of the following options are ***not*** available in Terraform OSS/CLI and Terraform Cloud (free)? (select three)

A. Sentinel (Correct)
B. Public Module Registry
C. Audit Logging (Correct)
D. Single Sign-On (SSO) (Correct)
E. Workspaces

Explanation
Single Sign-On requires Terraform Cloud for Business or Terraform Enterprise. It is NOT available in Terraform OSS or Terraform Cloud (free)

Sentinel is available in Terraform Cloud (Team & Governance), Terraform Enterprise, and Terraform Cloud for Business. It is NOT available in Terraform OSS or Terraform Cloud (free).

Audit Logging is available in Terraform Cloud (Team & Governance), Terraform Enterprise, and Terraform Cloud for Business. It is NOT available in Terraform OSS or Terraform Cloud (free).

Public Module Registry is available to users of any version of Terraform.

Workspaces are a feature of all versions of Terraform, both Terraform OSS/Cloud and all other paid versions.

This is a really good table that breaks down the features per Terraform offering.