Bhargav Bachina Follow

Jul 13, 2020 · 47 min read · ✦ · Listen

250 Practice Questions For Terraform Associate Certification

Read and Practice these questions before your exam

Terraform Certification

The Terraform Associate certification is for Cloud Engineers specializing in operations, IT, or developers who know the basic concepts and skills associated with open source HashiCorp Terraform. Candidates will be best prepared for this exam if they have professional experience using Terraform in production, but performing the exam objectives in a personal demo environment may also be sufficient.

Since this exam is multiple-choice, multiple-answer, and fill in the banks' questions, we need a lot of practice before the exam. This article helps you understand, practice,

and get you ready for the exam. ***All the questions and answers are taken straight from their documentation. These are only practice questions.***

We are not going to discuss any concepts here, rather, I just want to create a bunch of practice questions for this exam based on the curriculum provided here.

- *Understand infrastructure as code (IaC) concepts*
- *Understand Terraform's purpose (vs other IaC)*
- *Understand Terraform basics*
- *Use the Terraform CLI (outside of core workflow)*
- *Interact with Terraform modules*
- *Navigate Terraform workflow*
- *Implement and maintain state*
- *Read, generate, and modify the configuration*
- *Understand Terraform Cloud and Enterprise capabilities*

Understand infrastructure as code (IaC) concepts

Practice questions based on these concepts

- Explain what IaC is
- Describe the advantages of IaC patterns

1. What is Infrastructure as Code?

```
You write and execute the code to define, deploy, update, and destroy
your infrastructure
```

2. What are the benefits of IaC?

a. Automation
We can bring up the servers with one script and scale up and down based on our load with the same script.

b. Reusability of the code
We can reuse the same code

c. Versioning
We can check it into version control and we get versioning. Now we can see an incremental history of who changed what, how is our infrastructure actually defined at any given point of time, and wehave this transparency of documentation

IaC makes changes idempotent, consistent, repeatable, and predictable.

3. How using IaC make it easy to provision infrastructure?

IaC makes it easy to provision and apply infrastructure configurations, saving time. It standardizes workflows across different infrastructure providers (e.g., VMware, AWS, Azure, GCP, etc.) by using a common syntax across all of them.

4. What is Ideompodent in terms of IaC?

The idempotent characteristic provided by IaC tools ensures that, even if the same code is applied multiple times, the result remains the same.

5. What are Day 0 and Day 1 activities?

IaC can be applied throughout the lifecycle, both on the initial build, as well as throughout the life of the infrastructure. Commonly, these are referred to as Day 0 and Day 1 activities.

“Day 0” code provisions and configures your initial infrastructure.

“Day 1” refers to OS and application configurations you apply after you’ve initially built your infrastructure.

6. What are the use cases of Terraform?

```
Heroku App Setup
Multi-Tier Applications
Self-Service Clusters
Software Demos
Disposable Environments
Software Defined Networking
Resource Schedulers
Multi-Cloud Deployment
```

https://www.terraform.io/intro/use-cases.html

7. What are the advantages of Terraform?

```
Platform Agnostic
State Management
Operator Confidence
```

https://learn.hashicorp.com/terraform/getting-started/intro

8. Where do you describe all the components or your entire datacenter so that Terraform provision those?

```
Configuration files ends with *.tf
```

9. How can Terraform build infrastructure so efficiently?

```
Terraform builds a graph of all your resources, and parallelizes the
creation and modification of any non-dependent resources. Because of
this, Terraform builds infrastructure as efficiently as possible, and
operators get insight into dependencies in their infrastructure.
```

Understand Terraform's purpose (vs other IaC)

Practice questions based on these concepts

- Explain multi-cloud and provider-agnostic benefits
- Explain the benefits of state

10. What is multi-cloud deployment?

```
Provisoning your infrastrcutire into multiple cloud providers to
increase fault-tolerance of your applications.
```

11. How multi-cloud deployment is useful?

```
By using only a single region or cloud provider, fault tolerance is
limited by the availability of that provider.

Having a multi-cloud deployment allows for more graceful recovery of
the loss of a region or entire provider.
```

12. What is cloud-agnostic in terms of provisioning tools?

```
cloud-agnostic and allows a single configuration to be used to manage
multiple providers, and to even handle cross-cloud dependencies.
```

13. Is Terraform cloud-agostic?

```
Yes
```

14. What is the use of terraform being cloud-agnostic?

It simplifies management and orchestration, helping operators build large-scale multi-cloud infrastructures.

15. What is the Terraform State?

Every time you run Terraform, it records information about what infrastructure it created in a *Terraform state file*.

By default, when you run Terraform in the folder */some/folder*, Terraform creates the file */some/folder/terraform.tfstate*.

This file contains a custom JSON format that records a mapping from the Terraform resources in your configuration files to the representation of those resources in the real world.

16. What is the purpose of the Terraform State?

Mapping to the Real World
Terraform requires some sort of database to map Terraform config to the real world because you can't find the same functionality in every cloud provider. You need to have some kind of mechanism to be cloud-agnostic

Metadata
Terraform must also track metadata such as resource dependencies, pointer to the provider configuration that was most recently used with the resource in situations where multiple aliased providers are present.

Performance
When running a terraform plan, Terraform must know the current state of resources in order to effectively determine the changes that it needs to make to reach your desired configuration.

For larger infrastructures, querying every resource is too slow. Many cloud providers do not provide APIs to query multiple resources at once, and the round trip time for each resource is hundreds of milliseconds. So, Terraform stores a cache of the attribute values for all resources in the state. This is the most optional feature of Terraform state and is done only as a performance improvement.

Syncing
When two people works on the same file and doing some changes to the infrastructure. Its very important for everyone to be working with the

```
same state so that operations will be applied to the same remote
objects.
```

https://www.terraform.io/docs/state/purpose.html

17. What is the name of the terraform state file?

```
terraform.tfstate
```

Understand Terraform basics

Practice questions based on these concepts

- Handle Terraform and provider installation and versioning
- Describe the plug-in based architecture
- Demonstrate using multiple providers
- Describe how Terraform finds and fetches providers
- Explain when to use and not use provisioners and when to use local-exec or remote-exec

18. How do you install terraform on different OS?

```
// Mac OS
brew install terraform

// Windows
choco install terraform
```

https://learn.hashicorp.com/terraform/getting-started/install

19. How do you manually install terraform?

```
step 1: Download the zip fille
step 2: mv ~/Downloads/terraform /usr/local/bin/terraform
```

20. Where do you put terraform configurations so that you can configure some behaviors of Terraform itself?

The special terraform configuration block type is used to configure some behaviors of Terraform itself, such as requiring a minimum Terraform version to apply your configuration.

```
terraform {
  # ...
}
```

21. Only constants are allowed inside the terraform block. Is this correct?

Yes

Within a terraform block, only constant values can be used; arguments may not refer to named objects such as resources, input variables, etc, and may not use any of the Terraform language built-in functions.

22. What are the Providers?

A provider is a plugin that Terraform uses to translate the API interactions with the service. A provider is responsible for understanding API interactions and exposing resources. Because Terraform can interact with any API, you can represent almost any infrastructure type as a resource in Terraform.

https://www.terraform.io/docs/configuration/providers.html

23. How do you configure a Provider?

```
provider "google" {
  project = "acme-app"
```

```
  region  = "us-central1"
}
```

The name given in the block header ("google" in this example) is the name of the provider to configure. Terraform associates each resource type with a provider by taking the first word of the resource type name (separated by underscores), and so the "google" provider is assumed to be the provider for the resource type name google_compute_instance.

The body of the block (between { and }) contains configuration arguments for the provider itself. Most arguments in this section are specified by the provider itself; in this example both project and region are specific to the google provider.

24. What are the meta-arguments that are defined by Terraform itself and available for all `provider` blocks?

version: Constraining the allowed provider versions

alias: using the same provider with different configurations for different resources

25. What is Provider initialization and why do we need?

Each time a new provider is added to configuration -- either explicitly via a provider block or by adding a resource from that provider -- Terraform must initialize the provider before it can be used.

Initialization downloads and installs the provider's plugin so that it can later be executed.

26. How do you initialize any Provider?

Provider initialization is one of the actions of ***terraform init.*** Running this command will download and initialize any providers that are not already initialized.

27. When you run `terraform init` command, all the providers are installed in the current working directory. Is this true?

Providers downloaded by **terraform init** are only installed for the current working directory; other working directories can have their own installed provider versions.

Note that ***terraform init*** cannot automatically download providers that are not distributed by HashiCorp. See Third-party Plugins below for installation instructions.

28. How do you constrain the provider version?

To constrain the provider version as suggested, add a required_providers block inside a terraform block:

```
terraform {
  required_providers {
    aws = "~> 1.0"
  }
}
```

29. How do you upgrade to the latest acceptable version of the provider?

```
terraform init --upgrade
```

It upgrade to the latest acceptable version of each provider
This command also upgrades to the latest versions of all Terraform modules.

30. How many ways you can configure provider versions?

1. With required_providers blocks under terraform block

```
terraform {
  required_providers {
    aws = "~> 1.0"
  }
}
```

2. Provider version constraints can also be specified using a version argument within a provider block

```
provider {
  version= "1.0"
}
```

31. How do you configure Multiple Provider Instances?

alias

You can optionally define multiple configurations for the same provider, and select which one to use on a per-resource or per-module basis.

32. Why do we need Multiple Provider instances?

Some of the example scenarios:

a. multiple regions for a cloud platform
b. targeting multiple Docker hosts
c. multiple Consul hosts, etc.

33. How do we define multiple Provider configurations?

To include multiple configurations for a given provider, include multiple provider blocks with the same provider name, but set the alias meta-argument to an alias name to use for each additional configuration.

```
# The default provider configuration
provider "aws" {
  region = "us-east-1"
}

# Additional provider configuration for west coast region
provider "aws" {
  alias  = "west"
  region = "us-west-2"
}
```

34. How do you select alternate providers?

```
By default, resources use a default provider configuration inferred
from the first word of the resource type name. For example, a resource
of type aws_instance uses the default (un-aliased) aws provider
configuration unless otherwise stated.

resource "aws_instance" "foo" {
  provider = aws.west

  # ...
}
```

35. What is the location of the user plugins directory?

```
Windows                     %APPDATA%\terraform.d\plugins
All other systems           ~/.terraform.d/plugins
```

36. Third-party plugins should be manually installed. Is that true?

```
True
```

37. The command `terraform init` cannot install third-party plugins? True or false?

```
True

Install third-party providers by placing their plugin executables in
the user plugins directory. The user plugins directory is in one of
the following locations, depending on the host operating system

Once a plugin is installed, terraform init can initialize it normally.
You must run this command from the directory where the configuration
files are located.
```

38. What is the naming scheme for provider plugins?

```
terraform-provider-<NAME>_vX.Y.Z
```

39. What is the CLI configuration File?

The CLI configuration file configures per-user settings for CLI behaviors, which apply across all Terraform working directories.

It is named either ***.terraformrc*** or ***terraform.rc***

40. Where is the location of the CLI configuration File?

On Windows, the file must be named named **terraform.rc** and placed in the relevant user's **%APPDATA%** directory.

On all other systems, the file must be named **.terraformrc** (note the leading period) and placed directly in the home directory of the relevant user.

The location of the Terraform CLI configuration file can also be specified using the **TF_CLI_CONFIG_FILE** environment variable.

41. What is Provider Plugin Cache?

By default, terraform init downloads plugins into a subdirectory of the working directory so that each working directory is self-contained. As a consequence, if you have multiple configurations that use the same provider then a separate copy of its plugin will be downloaded for each configuration.

Given that provider plugins can be quite large (on the order of hundreds of megabytes), this default behavior can be inconvenient for those with slow or metered Internet connections.

Therefore Terraform optionally allows the use of a local directory as a shared plugin cache, which then allows each distinct plugin binary to be downloaded only once.

42. How do you enable Provider Plugin Cache?

To enable the plugin cache, use the plugin_cache_dir setting in the CLI configuration file.

plugin_cache_dir = "$HOME/.terraform.d/plugin-cache"

Alternatively, the **TF_PLUGIN_CACHE_DIR** environment variable can be used to enable caching or to override an existing cache directory within a particular shell session:

43. When you are using plugin cache you end up growing cache directory with different versions. Whose responsibility to clean it?

User

Terraform will never itself delete a plugin from the plugin cache once it's been placed there. Over time, as plugins are upgraded, the cache directory may grow to contain several unused versions which must be manually deleted.

44. Why do we need to initialize the directory?

When you create a new configuration — or check out an existing configuration from version control — you need to initialize the directory

```
// Example

provider "aws" {
  profile = "default"
  region  = "us-east-1"
}

resource "aws_instance" "example" {
  ami           = "ami-2757f631"
  instance_type = "t2.micro"
}
```

Initializing a configuration directory downloads and installs providers used in the configuration, which in this case is the aws provider. Subsequent commands will use local settings and data during initialization.

45. What is the command to initialize the directory?

```
terraform init
```

46. If different teams are working on the same configuration. How do you make files to have consistent formatting?

```
terraform fmt
```

This command automatically updates configurations in the current directory for easy readability and consistency.

47. If different teams are working on the same configuration. How do you make files to have syntactically valid and internally consistent?

```
terraform validate
```

This command will check and report errors within modules, attribute names, and value types.

Validate your configuration. If your configuration is valid, Terraform will return a success message.

48. What is the command to create infrastructure?

```
terraform apply
```

49. What is the command to show the execution plan and not apply?

```
terraform plan
```

50. How do you inspect the current state of the infrastructure applied?

```
terraform show
```

When you applied your configuration, Terraform wrote data into a file called terraform.tfstate. This file now contains the IDs and properties of the resources Terraform created so that it can manage or destroy those resources going forward.

51. If your state file is too big and you want to list the resources from your state. What is the command?

```
terraform state list
```

https://learn.hashicorp.com/terraform/getting-started/build#manually-managing-state

52. What is plug-in based architecture?

Defining additional features as plugins to your core platform or core application. This provides extensibility, flexibility and isolation

53. What are Provisioners?

If you need to do some initial setup on your instances, then provisioners let you upload files, run shell scripts, or install and trigger other software like configuration management tools, etc.

54. How do you define provisioners?

```
resource "aws_instance" "example" {
  ami           = "ami-b374d5a5"
  instance_type = "t2.micro"

  provisioner "local-exec" {
    command = "echo hello > hello.txt"
  }
}
```

Provisioner block within the resource block. Multiple provisioner blocks can be added to define multiple provisioning steps. Terraform

supports multiple provisioners

https://learn.hashicorp.com/terraform/getting-started/provision

55. What are the types of provisioners?

local-exec
remote-exec

56. What is a local-exec provisioner and when do we use it?

The local-exec provisioner executing a command locally on your machine running Terraform.

We use this when we need to do something on our local machine without needing any external URL

57. What is a remote-exec provisioner and when do we use it?

Another useful provisioner is remote-exec which invokes a script on a remote resource after it is created.

This can be used to run a configuration management tool, bootstrap into a cluster, etc.

58. Are provisioners runs only when the resource is created or destroyed?

Provisioners are only run when a resource is *created or destroyed*. Provisioners that are run while destroying are Destroy provisioners.

They are not a replacement for configuration management and changing the software of an already-running server, and are instead just meant as a way to bootstrap a server.

59. What do we need to use a remote-exec?

In order to use a remote-exec provisioner, you must choose an **ssh** or **winrm** connection in the form of a connection block within the provisioner.

Here is an example

```
provider "aws" {
  profile = "default"
  region  = "us-west-2"
}

resource "aws_key_pair" "example" {
  key_name   = "examplekey"
  public_key = file("~/.ssh/terraform.pub")
}

resource "aws_instance" "example" {
  key_name      = aws_key_pair.example.key_name
  ami           = "ami-04590e7389a6e577c"
  instance_type = "t2.micro"

connection {
    type        = "ssh"
    user        = "ec2-user"
    private_key = file("~/.ssh/terraform")
    host        = self.public_ip
  }

provisioner "remote-exec" {
    inline = [
      "sudo amazon-linux-extras enable nginx1.12",
      "sudo yum -y install nginx",
      "sudo systemctl start nginx"
    ]
  }
}
```

60. When terraform mark the resources are tainted?

If a resource successfully creates but fails during provisioning, Terraform will error and mark the resource as "tainted".

A resource that is tainted has been physically created, but can't be considered safe to use since provisioning failed.

61. You applied the infrastructure with terraform apply and you have some tainted resources. You run an execution plan now what happens to those tainted resources?

When you generate your next execution plan, Terraform will not attempt to restart provisioning on the same resource because it isn't guaranteed to be safe.

Instead, Terraform will remove any tainted resources and create new resources, attempting to provision them again after creation.

https://learn.hashicorp.com/terraform/getting-started/provision

62. Terraform also does not automatically roll back and destroy the resource during the apply when the failure happens. Why?

Terraform also does not automatically roll back and destroy the resource during the apply when the failure happens, because that would go against the execution plan: the execution plan would've said a resource will be created, but does not say it will ever be deleted. If you create an execution plan with a tainted resource, however, the plan will clearly state that the resource will be destroyed because it is tainted.

https://learn.hashicorp.com/terraform/getting-started/provision

63. How do you manually taint a resource?

```
terraform taint resource.id
```

64. Does the taint command modify the infrastructure?

```
terraform taint resource.id
```

This command will not modify infrastructure, but does modify the state file in order to mark a resource as tainted. Once a resource is marked as tainted, the next plan will show that the resource will be destroyed and recreated and the next apply will implement this change.

65. By default, provisioners that fail will also cause the Terraform apply itself to fail. Is this true?

True

66. By default, provisioners that fail will also cause the Terraform apply itself to fail. How do you change this?

The **on_failure** setting can be used to change this.

The allowed values are:

continue: Ignore the error and continue with creation or destruction.

fial: Raise an error and stop applying (the default behavior). If this is a creation provisioner, taint the resource.

```
// Example

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

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

67. How do you define destroy provisioner and give an example?

You can define destroy provisioner with the parameter when

```
provisioner "remote-exec" {
    when = "destroy"

    # <...snip...>

}
```

68. How do you apply constraints for the provider versions?

The required_providers setting is a map specifying a version constraint for each provider required by your configuration.

```
terraform {
  required_providers {
    aws = ">= 2.7.0"
  }
}
```

69. What should you use to set both a lower and upper bound on versions for each provider?

```
~>

terraform {
  required_providers {
    aws = "~> 2.7.0"
  }
}
```

70. How do you try experimental features?

```
In releases where experimental features are available, you can enable
them on a per-module basis by setting the experiments argument inside
a terraform block:

terraform {
  experiments = [example]
}
```

71. When does the terraform does not recommend using provisions?

```
Passing data into virtual machines and other compute resources

https://www.terraform.io/docs/provisioners/#passing-data-into-virtual-
machines-and-other-compute-resources

Running configuration management software

https://www.terraform.io/docs/provisioners/#running-configuration-
management-software
```

72. Expressions in `provisioner` blocks cannot refer to their parent resource by name. Is this true?

```
True

The self object represents the provisioner's parent resource, and has
all of that resource's attributes.

For example, use self.public_ip to reference an aws_instance's
public_ip attribute.
```

73. What does this symbol version = "~> 1.0" mean when defining versions?

```
Any version more than 1.0 and less than 2.0
```

74. Terraform supports both cloud and on-premises infrastructure platforms. Is this true?

```
True
```

75. Terraform assumes an empty default configuration for any provider that is not explicitly configured. A provider block can be empty. Is this true?

```
True
```

76. How do you configure the required version of Terraform CLI can be used with your configuration?

```
The required_version setting can be used to constrain which versions
of the Terraform CLI can be used with your configuration. If the
running version of Terraform doesn't match the constraints specified,
Terraform will produce an error and exit without taking any further
actions.
```

77. Terraform CLI versions and provider versions are independent of each other. Is this true?

```
True
```

*78. You are configuring aws provider and it is always recommended to hard code aws credentials in *.tf files. Is this true?*

```
False

HashiCorp recommends that you never hard-code credentials into *.tf
configuration files. We are explicitly defining the default AWS config
profile here to illustrate how Terraform should access sensitive
credentials.

If you leave out your AWS credentials, Terraform will automatically
search for saved API credentials (for example, in ~/.aws/credentials)
or IAM instance profile credentials. This is cleaner when .tf files
are checked into source control or if there is more than one admin
user
```

79. You are provisioning the infrastructure with the command terraform apply and you noticed one of the resources failed. How do you remove that resource without affecting the whole infrastructure?

```
You can taint the resource ans the next apply will destroy the
resource

terraform taint <resource.id>
```

Use the Terraform CLI (outside of core workflow)

Practice questions based on these concepts

- Given a scenario: choose when to use `terraform fmt` to format code

- Given a scenario: choose when to use `terraform taint` to taint Terraform resources
- Given a scenario: choose when to use `terraform import` to import existing infrastructure into your Terraform state
- Given a scenario: choose when to use `terraform workspace` to create workspaces
- Given a scenario: choose when to use `terraform state` to view Terraform state
- Given a scenario: choose when to enable verbose logging and what the outcome/value is

80. What is command fmt?

```
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.
```

81. What is the recommended approach after upgrading terraform?

```
The canonical format may change in minor ways between Terraform
versions, so after upgrading Terraform we recommend to proactively run
terraform fmt on your modules along with any other changes you are
making to adopt the new version.
```

82. What is the command usage?

```
terraform fmt [options] [DIR]
```

83. By default, `fmt` scans the current directory for configuration files. Is this true?

```
True
```

By default, fmt scans the current directory for configuration files. If the dir argument is provided then it will scan that given directory instead. If dir is a single dash (-) then fmt will read from standard input (STDIN).

84. You are formatting the configuration files and what is the flag you should use to see the differences?

```
terraform fmt -diff
```

85. You are formatting the configuration files and what is the flag you should use to process the subdirectories as well?

```
terraform fmt -recursive
```

86. You are formatting configuration files in a lot of directories and you don't want to see the list of file changes. What is the flag that you should use?

```
terraform fmt -list=false
```

87. What is the command taint?

The terraform ***taint*** command manually marks a Terraform-managed resource as tainted, forcing it to be destroyed and recreated on the next apply.

This command *will not* modify infrastructure, but does modify the state file in order to mark a resource as tainted. Once a resource is marked as tainted, the next plan will show that the resource will be destroyed and recreated and the next apply will implement this change.

88. What is the command usage?

```
terraform taint [options] address
```

The address argument is the address of the resource to mark as tainted. The address is in the resource address syntax syntax

89. When you are tainting a resource terraform reads the default state file terraform.tfstate. What is the flag you should use to read from a different path?

```
terraform taint -state=path
```

90. Give an example of tainting a single resource?

```
terraform taint aws_security_group.allow_all
```

```
The resource aws_security_group.allow_all in the module root has been marked as tainted.
```

91. Give an example of tainting a resource within a module?

```
terraform taint "module.couchbase.aws_instance.cb_node[9]"
```

```
Resource instance module.couchbase.aws_instance.cb_node[9] has been marked as tainted.
```

92. What is the command import?

The terraform import command is used to import existing resources into Terraform.

Terraform is able to import existing infrastructure. This allows you take resources you've created by some other means and bring it under Terraform management.

This is a great way to slowly transition infrastructure to Terraform, or to be able to be confident that you can use Terraform in the future if it potentially doesn't support every feature you need today.

93. What is the command import usage?

```
terraform import [options] ADDRESS ID
```

94. What is the default workspace name?

```
default
```

95. What are workspaces?

Each Terraform configuration has an associated backend that defines how operations are executed and where persistent data such as the Terraform state are stored.

The persistent data stored in the backend belongs to a *workspace*. Initially the backend has only one workspace, called "default", and thus there is only one Terraform state associated with that configuration.

Certain backends support *multiple* named workspaces, allowing multiple states to be associated with a single configuration.

96. What is the command to list the workspaces?

```
terraform workspace list
```

97. What is the command to create a new workspace?

```
terraform workspace new <name>
```

98. What is the command to show the current workspace?

```
terraform workspace show
```

99. What is the command to switch the workspace?

```
terraform workspace select <workspace name>
```

100. What is the command to delete the workspace?

```
terraform workspace delete <workspace name>
```

101. Can you delete the default workspace?

```
No. You can't ever delete default workspace
```

102. You are working on the different workspaces and you want to use a different number of instances based on the workspace. How do you achieve that?

```
resource "aws_instance" "example" {
  count = "${terraform.workspace == "default" ? 5 : 1}"

  # ... other arguments
}
```

103. You are working on the different workspaces and you want to use tags based on the workspace. How do you achieve that?

```
resource "aws_instance" "example" {
  tags = {
    Name = "web - ${terraform.workspace}"
  }
```

```
  # ... other arguments
}
```

104. You want to create a parallel, distinct copy of a set of infrastructure in order to test a set of changes before modifying the main production infrastructure. How do you achieve that?

```
Workspaces
```

105. What is the command state?

```
The terraform state command is used for advanced state management. As
your Terraform usage becomes more advanced, there are some cases where
you may need to modify the Terraform state. Rather than modify the
state directly, the terraform state commands can be used in many cases
instead.

https://www.terraform.io/docs/commands/state/index.html
```

106. What is the command usage?

```
terraform state <subcommand> [options] [args]
```

107. You are working on terraform files and you want to list all the resources. What is the command you should use?

```
terraform state list
```

108. How do you list the resources for the given name?

```
terraform state list <resource name>
```

109. What is the command that shows the attributes of a single resource in the state file?

```
terraform state show 'resource name'
```

110. How do you do debugging terraform?

Terraform has detailed logs which can be enabled by setting the **TF_LOG** environment variable to any value.

This will cause detailed logs to appear on stderr.

You can set **TF_LOG** to one of the log levels **TRACE**, **DEBUG**, **INFO**, **WARN** or **ERROR** to change the verbosity of the logs. **TRACE** is the most verbose and it is the default if **TF_LOG** is set to something other than a log level name.

To persist logged output you can set **TF_LOG_PATH** in order to force the log to always be appended to a specific file when logging is enabled.

Note that even when **TF_LOG_PATH** is set, **TF_LOG** must be set in order for any logging to be enabled.

https://www.terraform.io/docs/internals/debugging.html

111. If terraform crashes where should you see the logs?

crash.log

If Terraform ever crashes (a "panic" in the Go runtime), it saves a log file with the debug logs from the session as well as the panic message and backtrace to crash.log.

https://www.terraform.io/docs/internals/debugging.html

112. What is the first thing you should do when the terraform crashes?

panic message

The most interesting part of a crash log is the panic message itself and the backtrace immediately following. So the first thing to do is to search the file for panic

https://www.terraform.io/docs/internals/debugging.html

113. You are building infrastructure for different environments for example test and dev. How do you maintain separate states?

There are two primary methods to separate state between environments:

directories
workspaces

114. What is the difference between directory-separated and workspace-separated environments?

Directory separated environments rely on duplicate Terraform code, which may be useful if your deployments need differ, for example to test infrastructure changes in development. But they can run the risk of creating drift between the environments over time.

Workspace-separated environments use the same Terraform code but have different state files, which is useful if you want your environments to stay as similar to each other as possible, for example if you are providing development infrastructure to a team that wants to simulate running in production.

115. What is the command to pull the remote state?

```
terraform state pull
```

This command will download the state from its current location and output the raw format to stdout.

https://www.terraform.io/docs/commands/state/pull.html

116. What is the command is used manually to upload a local state file to a remote state

```
terraform state push
```

The terraform state push command is used to manually upload a local state file to remote state. This command also works with local state.

https://www.terraform.io/docs/commands/state/push.html

117. The command terraform taint modifies the state file and doesn't modify the infrastructure. Is this true?

True

This command *will not* modify infrastructure, but does modify the state file in order to mark a resource as tainted. Once a resource is marked as tainted, the next plan will show that the resource will be destroyed and recreated and the next apply will implement this change.

118. Your team has decided to use terraform in your company and you have existing infrastructure. How do you migrate your existing resources to terraform and start using it?

You should use terraform import and modify the infrastrcuture in the terraform files and do the terraform workflow (init, plan, apply)

119. When you are working with the workspaces how do you access the current workspace in the configuration files?

${terraform.workspace}

120. When you are using workspaces where does the Terraform save the state file for the local state?

terraform.tfstate.d

For local state, Terraform stores the workspace states in a directory called terraform.tfstate.d.

121. When you are using workspaces where does the Terraform save the state file for the remote state?

For remote state, the workspaces are stored directly in the configured backend.

122. How do you remove items from the Terraform state?

```
terraform state rm 'packet_device.worker'
```

The **terraform state rm** command is used to remove items from the Terraform state. This command can remove single resources, single instances of a resource, entire modules, and more.

https://www.terraform.io/docs/commands/state/rm.html

123. How do you move the state from one source to another?

```
terraform state mv 'module.app' 'module.parent.module.app'
```

The terraform state mv command is used to move items in a Terraform state. This command can move single resources, single instances of a resource, entire modules, and more. This command can also move items to a completely different state file, enabling efficient refactoring.

https://www.terraform.io/docs/commands/state/mv.html

124. How do you rename a resource in the terraform state file?

```
terraform state mv 'packet_device.worker' 'packet_device.helper'
```

The above example renames the packet_device resource named worker to helper:

Interact with Terraform modules

Practice questions based on these concepts

- Contrast module source options
- Interact with module inputs and outputs
- Describe variable scope within modules/child modules
- Discover modules from the public Terraform Module Registry
- Defining module version

125. Where do you find and explore terraform Modules?

The Terraform Registry makes it simple to find and use modules.

The search query will look at module name, provider, and description to match your search terms. On the results page, filters can be used further refine search results.

126. How do you make sure that modules have stability and compatibility?

By default, only verified modules are shown in search results.

Verified modules are reviewed by HashiCorp to ensure stability and compatibility.

By using the filters, you can view unverified modules as well.

127. How do you download any modules?

You need to add any module in the configuration file like below

```
module "consul" {
  source = "hashicorp/consul/aws"
  version = "0.1.0"
}
```

terraform init command will download and cache any modules referenced by a configuration.

128. What is the syntax for referencing a registry module?

```
<NAMESPACE>/<NAME>/<PROVIDER>

// for example
module "consul" {
  source = "hashicorp/consul/aws"
  version = "0.1.0"
}
```

129. What is the syntax for referencing a private registry module?

```
<HOSTNAME>/<NAMESPACE>/<NAME>/<PROVIDER>

// for example
module "vpc" {
  source = "app.terraform.io/example_corp/vpc/aws"
  version = "0.9.3"
}
```

130. The terraform recommends that all modules must follow semantic versioning. Is this true?

```
True
```

131. What is a Terraform Module?

```
A Terraform module is a set of Terraform configuration files in a
single directory. Even a simple configuration consisting of a single
directory with one or more .tf files is a module.
```

132. Why do we use modules for?

* Organize configuration
* Encapsulate configuration
* Re-use configuration
* Provide consistency and ensure best practices

https://learn.hashicorp.com/terraform/modules/modules-overview

133. How do you call modules in your configuration?

Your configuration can use module blocks to call modules in other directories.

When Terraform encounters a module block, it loads and processes that module's configuration files.

134. How many ways you can load modules?

Local and remote modules

Modules can either be loaded from the local filesystem, or a remote source.

Terraform supports a variety of remote sources, including the Terraform Registry, most version control systems, HTTP URLs, and Terraform Cloud or Terraform Enterprise private module registries.

135. What are the best practices for using Modules?

1. Start writing your configuration with modules in mind. Even for modestly complex Terraform configurations managed by a single person, you'll find the benefits of using modules outweigh the time it takes to use them properly.

2. Use local modules to organize and encapsulate your code. Even if you aren't using or publishing remote modules, organizing your configuration in terms of modules from the beginning will significantlty reduce the burden of maintaining and updating your configuration as your infrastructure grows in complexity.

3. Use the public Terraform Registry to find useful modules. This way you can more quickly and confidently implement your configuration by

relying on the work of others to implement common infrastructure scenarios.

4. Publish and share modules with your team. Most infrastructure is managed by a team of people, and modules are important way that teams can work together to create and maintain infrastructure. As mentioned earlier, you can publish modules either publicly or privately. We will see how to do this in a future guide in this series.

https://learn.hashicorp.com/terraform/modules/modules-overview#module-best-practices

136. What are the different source types for calling modules?

Local paths
Terraform Registry
GitHub
Generic Git, Mercurial repositories
Bitbucket
HTTP URLs
S3 buckets
GCS buckets

https://www.terraform.io/docs/modules/sources.html

137. What are the arguments you need for using modules in your configuration?

source and version

```
// example
module "consul" {
  source = "hashicorp/consul/aws"
  version = "0.1.0"
}
```

138. How do you set input variables for the modules?

The configuration that calls a module is responsible for setting its input values, which are passed as arguments in the module block. Aside from source and version, most of the arguments to a module block will set variable values.

> On the Terraform registry page for the AWS VPC module, you will see an Inputs tab that describes all of the input variables that module supports.

For example, we have defined a lot of input variables for the modules such as ads, cidr, name, etc

main.tf

139. How do you access output variables from the modules?

You can access them by referring

module.<MODULE NAME>.<OUTPUT NAME>

140. Where do you put output variables in the configuration?

Module outputs are usually either passed to other parts of your configuration, or defined as outputs in your root module. You will see both uses in this guide.

Inside your configuration's directory, **outputs.tf** will need to contain:

outputs.tf

141. How do you pass input variables in the configuration?

You can define **variables.tf** in the root folder

```
variable "vpc_name" {
  description = "Name of VPC"
  type        = string
  default     = "example-vpc"
}
```

Then you can access these varibles in the configuration like this

```
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
}
```

142. What is the child module?

A module that is called by another configuration is sometimes referred to as a "child module" of that configuration.

143. When you use local modules you don't have to do the command init or get every time there is a change in the local module. why?

When installing a local module, Terraform will instead refer directly to the source directory.

```
Because of this, Terraform will automatically notice changes to local
modules without having to re-run terraform init or terraform get.
```

144. When you use remote modules what should you do if there is a change in the module?

```
When installing a remote module, Terraform will download it into the
.terraform directory in your configuration's root directory.

You should initialize with terraform init
```

145. A simple configuration consisting of a single directory with one or more .tf files is a module. Is this true?

```
True
```

146. When using a new module for the first time, you must run either `terraform init` or `terraform get` to install the module. Is this true?

```
True
```

147. When installing the modules and where does the terraform save these modules?

```
.terraform/modules

// Example

.terraform/modules
├── ec2_instances
│   └── terraform-aws-modules-terraform-aws-ec2-instance-ed6dcd9
├── modules.json
└── vpc
    └── terraform-aws-modules-terraform-aws-vpc-2417f60
```

148. What is the required argument for the module?

source

All modules require a source argument, which is a meta-argument defined by Terraform CLI. Its value is either the path to a local directory of the module's configuration files, or a remote module source that Terraform should download and use. This value must be a literal string with no template sequences; arbitrary expressions are not allowed. For more information on possible values for this argument, see Module Sources.

149. What are the other optional meta-arguments along with the source when defining modules

version - (Optional) A version constraint string that specifies which versions of the referenced module are acceptable. The newest version matching the constraint will be used. version is supported only for modules retrieved from module registries.

providers - (Optional) A map whose keys are provider configuration names that are expected by child module and whose values are corresponding provider names in the calling module. This allows provider configurations to be passed explicitly to child modules. If not specified, the child module inherits all of the default (un-aliased) provider configurations from the calling module.

Navigate Terraform workflow

Practice questions based on these concepts

- Describe Terraform workflow (Write -> Plan -> Create)
- Initialize a Terraform working directory (terraform init)
- Validate a Terraform configuration (terraform validate)
- Generate and review an execution plan for Terraform (terraform plan)
- Execute changes to infrastructure with Terraform (terraform apply)
- Destroy Terraform managed infrastructure (terraform destroy)

150. What is the Core Terraform workflow?

The core Terraform workflow has three steps:

1. **Write** - Author infrastructure as code.
2. **Plan** - Preview changes before applying.
3. **Apply** - Provision reproducible infrastructure.

151. What is the workflow when you work as an Individual Practitioner?

https://www.terraform.io/guides/core-workflow.html#working-as-an-individual-practitioner

152. What is the workflow when you work as a team?

https://www.terraform.io/guides/core-workflow.html#working-as-a-team

153. What is the workflow when you work as a large organization?

https://www.terraform.io/guides/core-workflow.html#the-core-workflow-enhanced-by-terraform-cloud

154. What is the command init?

The terraform ***init*** command is used to initialize a working directory containing Terraform configuration files.

This is the first command that should be run after writing a new Terraform configuration or cloning an existing one from version control.

It is safe to run this command multiple times.

155. You recently joined a team and you cloned a terraform configuration files from the version control system. What is the first command you should use?

terraform init

This command performs several different initialization steps in order to prepare a working directory for use.

This command is always safe to run multiple times, to bring the working directory up to date with changes in the configuration.

Though subsequent runs may give errors, this command will never delete your existing configuration or state.

If no arguments are given, the configuration in the current working directory is initialized. It is recommended to run Terraform with the current working directory set to the root directory of the configuration, and omit the DIR argument.

https://www.terraform.io/docs/commands/init.html

156. What is the flag you should use to upgrade modules and plugins a part of their respective installation steps?

upgrade

```
terraform init -upgrade
```

157. When you are doing initialization with terraform init, you want to skip backend initialization. What should you do?

```
terraform init -backend=false
```

158. When you are doing initialization with terraform init, you want to skip child module installation. What should you do?

```
terraform init -get=false
```

159. When you are doing initialization where do all the plugins stored?

```
On most operationg systems :          ~/.terraform.d/plugins
on Windows                 :          %APPDATA%\terraform.d\plugins
```

160. When you are doing initialization with terraform init, you want to skip plugin installation. What should you do?

```
terraform init -get-plugins=false
```

Skips plugin installation. Terraform will use plugins installed in the user plugins directory, and any plugins already installed for the current working directory. If the installed plugins aren't sufficient for the configuration, init fails.

161. What does the command terraform validate does?

The ***terraform validate*** command validates the configuration files in a directory, referring only to the configuration and not accessing any remote services such as remote state, provider APIs, etc.

Validate runs checks that verify whether a configuration is syntactically valid and internally consistent, regardless of any provided variables or existing state.

It is thus primarily useful for general verification of reusable modules, including correctness of attribute names and value types.

https://www.terraform.io/docs/commands/validate.html

162. What does the command plan do?

The ***terraform plan*** command is used to create an execution plan. Terraform performs a refresh, unless explicitly disabled, and then determines what actions are necessary to achieve the desired state specified in the configuration files.

163. What does the command apply do?

The ***terraform apply*** command is used to apply the changes required to reach the desired state of the configuration, or the pre-determined set of actions generated by a terraform plan execution plan.

https://www.terraform.io/docs/commands/apply.html

164. You are applying the infrastructure with the command apply and you don't want to do interactive approval. Which flag should you use?

```
terraform apply -auto-approve
```

https://www.terraform.io/docs/commands/apply.html

165. What does the command destroy do?

The ***terraform destroy*** command is used to destroy the Terraform-managed infrastructure.

166. How do you preview the behavior of the command terraform destroy?

```
terraform plan -destroy
```

167. What are implicit and explicit dependencies?

Implicit dependency:
By studying the resource attributes used in interpolation expressions, Terraform can automatically infer when one resource depends on another.

Terraform uses this dependency information to determine the correct order in which to create the different resources.

Implicit dependencies via interpolation expressions are the primary way to inform Terraform about these relationships, and should be used

whenever possible.

Explicit dependency:
Sometimes there are dependencies between resources that are *not* visible to Terraform. The **depends_on** argument is accepted by any resource and accepts a list of resources to create *explicit dependencies* for.

168. Give an example of implicit dependency?

In the example below, the reference to **aws_instance.example.id** creates an ***implicit*** *dependency* on the **aws_instance named example**.

```
provider "aws" {
  profile    = "default"
  region     = "us-east-1"
}

resource "aws_instance" "example" {
  ami           = "ami-b374d5a5"
  instance_type = "t2.micro"
}

resource "aws_eip" "ip" {
    vpc = true
    instance = aws_instance.example.id
}
```

169. Give an example of explicit dependency?

In the example below, an application we will run on our EC2 instance expects to use a specific Amazon S3 bucket, but that dependency is configured inside the application code and thus not visible to Terraform. In that case, we can use depends_on to explicitly declare the dependency

```
resource "aws_s3_bucket" "example" {
  bucket = "some_bucket"
  acl    = "private"
}

resource "aws_instance" "example" {
  ami           = "ami-2757f631"
  instance_type = "t2.micro"
```

```
  depends_on = [aws_s3_bucket.example]
}
```

170. How do you save the execution plan?

```
terraform plan -out=tfplan

you can use that file with apply

terraform apply tfplan
```

171. You have started writing terraform configuration and you are using some sample configuration as a basis. How do you copy the example configuration into your working directory?

```
terraform init -from-module=MODULE-SOURCE

https://www.terraform.io/docs/commands/init.html#copy-a-source-module
```

172. What is the flag you should use with the terraform plan to get detailed on the exit codes?

```
terraform plan -detailed-exitcode

Return a detailed exit code when the command exits. When provided,
this argument changes the exit codes and their meanings to provide
more granular information about what the resulting plan contains:

* 0 = Succeeded with empty diff (no changes)
* 1 = Error
* 2 = Succeeded with non-empty diff (changes present)
```

173. How do you target only specific resources when you run a terraform plan?

```
-target=resource - A Resource Address to target. This flag can be used
multiple times. See below for more information.
```

174. How do you update the state prior to checking differences when you run a terraform plan?

```
terraform plan -refresh=true
```

175. The behavior of any `terraform destroy` *command can be previewed at any time with an equivalent* `terraform plan -destroy` *command. Is this true?*

```
True
```

176. You have the following file and created two resources docker_image and docker_container with the command `terraform apply` *and you go to the terminal and delete the container with the command* `docker rm`*. You come back to your configuration and run the command again. Does terraform recreates the resource?*

main.tf

```
Yes. Terrsform creates the resource again since the execution plan
says two resources and the terraform always maintains the desired
state
```

177. You created a VM instance on AWS cloud provider with the terraform configuration and you log in AWS console and removed the instance. What does the next apply do?

```
It creates the instance again
```

178. You have the following file and created two resources docker_image and docker_container with the command `terraform plan` *and you go to the terminal and delete the container with the command* `docker rm`*. You come back to your configuration and run the command again. What is the output of the command plan?*

```
Terraform will perform the following actions:

# docker_container.nginx will be created

Plan: 1 to add, 0 to change, 0 to destroy.
```

Implement and maintain state

Practice questions based on these concepts

- Describe default local backend
- Outline state locking
- Handle backend authentication methods
- Describe remote state storage mechanisms and supported standard backends
- Describe the effect of Terraform refresh on state
- Describe backend block in configuration and best practices for partial configurations
- Understand secret management in state files

179. What are Backends?

```
A "backend" in Terraform determines how state is loaded and how an
operation such as apply is executed. This abstraction enables non-
local file state storage, remote execution, etc.

By default, Terraform uses the "local" backend, which is the normal
behavior of Terraform
```

180. What is local Backend?

```
The local backend stores state on the local filesystem, locks that
state using system APIs, and performs operations locally.

// Example

terraform {
  backend "local" {
    path = "relative/path/to/terraform.tfstate"
  }
}
```

181. What is the default path for the local backend?

This defaults to "terraform.tfstate" relative to the root module by default.

182. What is State Locking?

If supported by your backend, Terraform will lock your state for all operations that could write state. This prevents others from acquiring the lock and potentially corrupting your state.

State locking happens automatically on all operations that could write state. You won't see any message that it is happening. If state locking fails, Terraform will not continue.

183. Does Terraform continue if state locking fails?

No.

If state locking fails, Terraform will not continue.

184. Can you disable state locking?

Yes.

You can disable state locking for most commands with the -lock flag but it is not recommended.

185. What are the types of Backend?

Standard: State management, functionality covered in State Storage & Locking

Enhanced: Everything in standard plus remote operations.

186. What are remote Backends?

Remote backends allow Terraform to use a shared storage space for state data, so any member of your team can use Terraform to manage the same infrastructure.

187. What is the benefit of using remote backend?

Remote state storage makes collaboration easier and keeps state and secret information off your local disk.

Remote state is loaded only in memory when it is used.

188. If you want to switch from using remote backend to local backend. What should you do?

If you want to move back to local state, you can remove the backend configuration block from your configuration and run ***terraform init again.***

Terraform will once again ask if you want to migrate your state back to local.

189. What does the command refresh do?

The terraform refresh command is used to reconcile the state Terraform knows about (via its state file) with the real-world infrastructure.

This can be used to detect any drift from the last-known state, and to update the state file.

190. Does the command refresh modify the infrastructure?

The command ***refresh*** does not modify infrastructure, but does modify the state file.

If the state is changed, this may cause changes to occur during the next plan or apply.

191. How do you backup the state to the remote backend?

1. When configuring a backend for the first time (moving from no defined backend to explicitly configuring one), Terraform will give you the option to migrate your state to the new backend. This lets you adopt backends without losing any existing state.

2. To be extra careful, we always recommend manually backing up your state as well. You can do this by simply copying your terraform.tfstate file to another location.

192. What is a partial configuration in terms of configuring Backends?

You do not need to specify every required argument in the backend configuration. Omitting certain arguments may be desirable to avoid storing secrets, such as access keys, within the main configuration. When some or all of the arguments are omitted, we call this a ***partial configuration.***

193. What are the ways to provide remaining arguments when using partial configuration?

Interactively: Terraform will interactively ask you for the required values, unless interactive input is disabled. Terraform will not prompt for optional values.

File: A configuration file may be specified via the init command line. To specify a file, use the -backend-config=PATH option when running terraform init. If the file contains secrets it may be kept in a secure data store, such as Vault, in which case it must be downloaded to the local disk before running Terraform.

Command-line key/value pairs: Key/value pairs can be specified via the init command line. Note that many shells retain command-line flags in a history file, so this isn't recommended for secrets. To specify a single key/value pair, use the -backend-config="KEY=VALUE" option when running terraform init.

https://www.terraform.io/docs/backends/config.html

194. What is the basic requirement when using partial configuration?

When using partial configuration, Terraform requires at a minimum that an empty backend configuration is specified in one of the root Terraform configuration files, to specify the backend type

```
// Example
terraform {
  backend "consul" {}
}
```

195. Give an example of passing partial configuration with Command-line Key/Value pairs?

```
terraform init \
    -backend-config="address=demo.consul.io" \
    -backend-config="path=example_app/terraform_state" \
    -backend-config="scheme=https"
```

196. How to unconfigure a backend?

If you no longer want to use any backend, you can simply remove the configuration from the file. Terraform will detect this like any other change and prompt you to reinitialize.

As part of the reinitialization, Terraform will ask if you'd like to migrate your state back down to normal local state. Once this is complete then Terraform is back to behaving as it does by default.

197. How do you encrypt sensitive data in the state?

Terraform Cloud always encrypts state at rest and protects it with TLS in transit. Terraform Cloud also knows the identity of the user requesting state and maintains a history of state changes. This can be used to control access and track activity. Terraform Enterprise also supports detailed audit logging.

The S3 backend supports encryption at rest when the encrypt option is enabled. IAM policies and logging can be used to identify any invalid access. Requests for the state go over a TLS connection.

198. Backends are completely optional. Is this true?

Backends are completely optional. You can successfully use Terraform without ever having to learn or use backends. However, they do solve pain points that afflict teams at a certain scale. If you're an individual, you can likely get away with never using backends.

199. What are the benefits of Backends?

Working in a team: Backends can store their state remotely and protect that state with locks to prevent corruption. Some backends such as Terraform Cloud even automatically store a history of all state revisions.

Keeping sensitive information off disk: State is retrieved from backends on demand and only stored in memory. If you're using a backend such as Amazon S3, the only location the state ever is persisted is in S3.

Remote operations: For larger infrastructures or certain changes, terraform apply can take a long, long time. Some backends support remote operations which enable the operation to execute remotely. You can then turn off your computer and your operation will still complete. Paired with remote state storage and locking above, this also helps in team environments.

200. Why should you be very careful with the Force unlocking the state?

Terraform has a force-unlock command to manually unlock the state if unlocking failed.

Be very careful with this command. If you unlock the state when someone else is holding the lock it could cause multiple writers. Force unlock should only be used to unlock your own lock in the situation where automatic unlocking failed.

To protect you, the force-unlock command requires a unique lock ID. Terraform will output this lock ID if unlocking fails. This lock ID acts as a nonce, ensuring that locks and unlocks target the correct lock.

201. You should only use force unlock command when automatic unlocking fails. Is this true?

```
True
```

Read, generate, and modify the configuration

Practice questions based on these concepts

- Demonstrate the use of variables and outputs
- Describe secure secret injection best practice
- Understand the use of the collection and structural types
- Create and differentiate resource and data configuration
- Use resource addressing and resource parameters to connect resources together
- Use Terraform built-in functions to write configuration
- Configure resource using a dynamic block
- Describe built-in dependency management (order of execution based)

202. How do you define a variable?

```
variable "region" {
  default = "us-east-1"
}
```

```
This defines the region variable within your Terraform configuration.
```

203. How do you access the variable in the configuration?

```
// accessing a variable

provider "aws" {
  region = var.region
}
```

204. How many ways you can assign variables in the configuration?

Command-line flags

```
terraform apply -var 'region=us-east-1'
```

From a file

To persist variable values, create a file and assign variables within this file. Create a file named terraform.tfvars with the following contents:

```
region = "us-east-1"
```

```
terraform apply \
  -var-file="secret.tfvars" \
  -var-file="production.tfvars"
```

From environment varibles

Terraform will read environment variables in the form of **TF_VAR_name** to find the value for a variable. For example, the **TF_VAR_region** variable can be set in the shell to set the region variable in Terraform.

UI input
If you execute terraform apply with any variable unspecified, Terraform will ask you to input the values interactively. These values are not saved, but this provides a convenient workflow when getting started with Terraform. UI input is not recommended for everyday use of Terraform.

205. Does environment variables support List and map types?

No

Environment variables can only populate string-type variables. List and map type variables must be populated via one of the other mechanisms.

206. How do you provision infrastructure in a staging environment or a production environment using the same Terraform configuration?

```
You can use different varible files with the same configuration

// Example

// For development
terraform apply -var-file="dev.tfvars"

// For test
terraform apply -var-file="test.tfvars"
```

207. How do you assign default values to variables?

```
If no value is assigned to a variable via any of these methods and the
variable has a default key in its declaration, that value will be used
for the variable.

variable "region" {
  default = "us-east-1"
}
```

208. What are the data types for the variables?

```
string
number
bool

list(<TYPE>)
set(<TYPE>)
map(<TYPE>)
object({<ATTR NAME> = <TYPE>, ... })
tuple([<TYPE>, ...])
```

209. Give an example of data type List variables?

```
Lists are defined either explicitly or implicitly.
```

```
variable "availability_zone_names" {
  type    = list(string)
  default = ["us-west-1a"]
}
```

210. Give an example of data type Map variables?

```
variable "region" {}
variable "amis" {
  type = map(string)
}

amis = {
  "us-east-1" = "ami-abc123"
  "us-west-2" = "ami-def456"
}

// accessing
resource "aws_instance" "example" {
  ami           = var.amis[var.region]
  instance_type = "t2.micro"
}
```

211. What is the Variable Definition Precedence?

The above mechanisms for setting variables can be used together in any combination. If the same variable is assigned multiple values, Terraform uses the *last* value it finds, overriding any previous values. Note that the same variable cannot be assigned multiple values within a single source.

Terraform loads variables in the following order, with later sources taking precedence over earlier ones:

* Environment variables

* The **terraform.tfvars** file, if present.

* The **terraform.tfvars.json** file, if present.

* Any ***.auto.tfvars or *.auto.tfvars.json** files, processed in lexical order of their filenames.

* Any -var and -var-file options on the command line, in the order they are provided. (This includes variables set by a Terraform Cloud workspace.)

212. What are the output variables?

output variables as a way to organize data to be easily queried and shown back to the Terraform user.

Outputs are a way to tell Terraform what data is important. This data is outputted when apply is called, and can be queried using the ***terraform output*** command.

213. Hoe do you define an output variable?

```
output "ip" {
  value = aws_eip.ip.public_ip
}
```

Multiple output blocks can be defined to specify multiple output variables.

214. How do you view outputs and queries them?

You will see the output when you run the following command
terraform apply

You can query the output with the following command
terraform output ip

215. What are the dynamic blocks?

some resource types include repeatable *nested blocks* in their arguments, which do not accept expressions

You can dynamically construct repeatable nested blocks like setting using a special dynamic block type, which is supported inside **resource, data, provider, and provisioner** blocks:

A dynamic block acts much like a for expression, but produces nested blocks instead of a complex typed value. It iterates over a given complex value, and generates a nested block for each element of that complex value.

https://www.terraform.io/docs/configuration/expressions.html#dynamic-blocks

example using dynamic blocks

216. What are the best practices for dynamic blocks?

Overuse of dynamic blocks can make configuration hard to read and maintain, so we recommend using them only when you need to hide

details in order to build a clean user interface for a re-usable module.

Always write nested blocks out literally where possible.

217. What are the Built-in Functions?

The Terraform language includes a number of built-in functions that you can call from within expressions to transform and combine values.

```
max(5, 12, 9)
```

218. Does Terraform language support user-defined functions?

No

The Terraform language does not support user-defined functions, and so only the functions built in to the language are available for use.

219. What is the built-in function to change string to a number?

parseint parses the given string as a representation of an integer in the specified base and returns the resulting number. The base must be between 2 and 62 inclusive.

```
> parseint("100", 10)
100
```

More Number Functions here
https://www.terraform.io/docs/configuration/functions/abs.html

220. What is the built-in function to evaluates given expression and returns a boolean whether the expression produced a result without any errors?

can

```
condition     = can(formatdate("", var.timestamp))
```

https://www.terraform.io/docs/configuration/functions/can.html

221. What is the built-in function to evaluates all of its argument expressions in turn and returns the result of the first one that does not produce any errors?

```
try

locals {
  example = try(
    [tostring(var.example)],
    tolist(var.example),
  )
}
```

222. What is Resource Address?

A **Resource Address** is a string that references a specific resource in a larger infrastructure. An address is made up of two parts:

```
[module path][resource spec]
```

223. What is the Module path?

A module path addresses a module within the tree of modules. It takes the form:

```
module.A.module.B.module.C...
```

Multiple modules in a path indicate nesting. If a module path is specified without a resource spec, the address applies to every resource within the module. If the module path is omitted, this addresses the root module.

224. What is the Resource spec?

A resource spec addresses a specific resource in the config. It takes the form:

resource_type.resource_name[resource index]

* resource type - Type of the resource being addressed.

* resource_name - User-defined name of the resource.

* [resource index] - an optional index into a resource with multiple instances, surrounded by square brace characters ([and]).

// Examples

```
resource "aws_instance" "web" {
  # ...
  count = 4
}

aws_instance.web[3]  // Refers to only last instance
aws_instance.web     // Refers to all four "web" instances.

resource "aws_instance" "web" {
  # ...
  for_each = {
    "terraform": "value1",
    "resource":  "value2",
    "indexing":  "value3",
    "example":   "value4",
  }
}

aws_instance.web["example"] // Refers to only the "example" instance in the config.
```

225. What are complex types and what are the collection types Terraform supports?

A *complex* type is a type that groups multiple values into a single value.

There are two categories of complex types:
collection types (for grouping similar values)

* list(...): a sequence of values identified by consecutive whole numbers starting with zero.

* map(...): a collection of values where each is identified by a string label.

* set(...): a collection of unique values that do not have any secondary identifiers or ordering.

structural types (for grouping potentially dissimilar values).

* object(...): a collection of named attributes that each have their own type.

* tuple(...): a sequence of elements identified by consecutive whole numbers starting with zero, where each element has its own type.

226. What are the named values available and how do we refer to?

Terraform makes several kinds of named values available. Each of these names is an expression that references the associated value; you can use them as standalone expressions, or combine them with other expressions to compute new values.

* <RESOURCE TYPE>.<NAME> is an object representing a managed resource of the given type and name. The attributes of the resource can be accessed using dot or square bracket notation.

* var.<NAME> is the value of the input variable of the given name.

* local.<NAME> is the value of the local value of the given name.

* module.<MODULE NAME>.<OUTPUT NAME> is the value of the specified output value from a child module called by the current module.

* data.<DATA TYPE>.<NAME> is an object representing a data resource of the given data source type and name. If the resource has the count argument set, the value is a list of objects representing its instances. If the resource has the for_each argument set, the value is a map of objects representing its instances.

* path.module is the filesystem path of the module where the expression is placed.

* path.root is the filesystem path of the root module of the configuration.

* path.cwd is the filesystem path of the current working directory. In normal use of Terraform this is the same as path.root, but some advanced uses of Terraform run it from a directory other than the root module directory, causing these paths to be different.

* terraform.workspace is the name of the currently selected workspace.

227. What is the built-in function that reads the contents of a file at the given path and returns them as a base64-encoded string?

```
filebase64(path)
```

https://www.terraform.io/docs/configuration/functions/filebase64.html

228. What is the built-in function that converts a timestamp into a different time format?

```
formatdate(spec, timestamp)
```

https://www.terraform.io/docs/configuration/functions/formatdate.html

229. What is the built-in function encodes a given value to a string using JSON syntax?

```
jsonencode({"hello"="world"})
```

https://www.terraform.io/docs/configuration/functions/jsonencode.html

230. What is the built-in function that calculates a full host IP address for a given host number within a given IP network address prefix?

```
> cidrhost("10.12.127.0/20", 16)
10.12.112.16
```

https://www.terraform.io/docs/configuration/functions/cidrhost.html

Understand Terraform Cloud and Enterprise capabilities

Practice questions based on these concepts

- Describe the benefits of Sentinel, registry, and workspaces
- Differentiate OSS and Terraform Cloud workspaces
- Summarize features of Terraform Cloud

231. What is Sentinel?

Sentinel is an embedded policy-as-code framework integrated with the HashiCorp Enterprise products. It enables fine-grained, logic-based policy decisions, and can be extended to use information from external sources.

232. What is the benefit of Sentinel?

Codifying policy removes the need for ticketing queues, without sacrificing enforcement.

One of the other benefits of Sentinel is that it also has a full testing framework.

Avoiding a ticketing workflow allows organizations to provide more self-service capabilities and end-to-end automation, minimizing the friction for developers and operators.

https://www.hashicorp.com/blog/why-policy-as-code/

233. What is the Private Module Registry?

Terraform Cloud's private module registry helps you share Terraform modules across your organization. It includes support for module versioning, a searchable and filterable list of available modules, and a configuration designer to help you build new workspaces faster.

234. What is the difference between public and private module registries when defined source?

The public registry uses a three-part ***<NAMESPACE>/<MODULE NAME>/<PROVIDER>*** format

private modules use a four-part ***<HOSTNAME>/<ORGANIZATION>/<MODULE NAME>/<PROVIDER>*** format

```
// example

module "vpc" {
  source  = "app.terraform.io/example_corp/vpc/aws"
```

```
  version = "1.0.4"
}
```

235. Where is the Terraform Module Registry available at?

https://registry.terraform.io/

236. What is a workspace?

A workspace contains everything Terraform needs to manage a given collection of infrastructure, and separate workspaces function like completely separate working directories.

237. What are the benefits of workspaces?

https://www.hashicorp.com/resources/terraform-enterprise-understanding-workspaces-and-modules/

238. You are configuring a remote backend in the terraform cloud. You didn't create an organization before you do terraform init. Does it work?

While the organization defined in the backend stanza **must** already exist,

239. You are configuring a remote backend in the terraform cloud. You didn't create a workspace before you do terraform init. Does it work?

Terraform Cloud will create it if necessary. If you opt to use a workspace that already exists, the workspace must not have any existing states.

240. Terraform workspaces when you are working with CLI and Terraform workspaces in the Terraform cloud. Is this correct?

If you are familiar with running Terraform using the CLI, you may have used Terraform workspaces. Terraform Cloud workspaces behave differently than Terraform CLI workspaces. Terraform CLI workspaces allow multiple state files to exist within a single directory, enabling you to use one configuration for multiple environments. Terraform Cloud workspaces contain everything needed to manage a given set of infrastructure, and function like separate working directories.

241. How do you authenticate the CLI with the terraform cloud?

Newer Versions:

1. terraform login
2. it will open the terraform cloud and generate the token
3. paste that token back in the CLI

https://learn.hashicorp.com/terraform/tfc/tfc_login

Older versions:

keep the following token in the CLI configuration file

```
credentials "app.terraform.io" {
  token = "xxxxxx.atlasv1.zzzzzzzzzzzzz"
}
```

https://www.terraform.io/docs/commands/cli-config.html#credentials

242. You are building infrastructure on your local machine and you changed your backend to remote backend with the Terraform cloud. What should you do to migrate the state to the remote backend?

```
terraform init
```

Once you have authenticated the remote backend, you're ready to migrate your local state file to Terraform Cloud. To begin the migration, reinitialize. This causes Terraform to recognize your changed backend configuration.

During reinitialization, Terraform presents a prompt saying that it will copy the state file to the new backend. Enter "yes" and Terraform will migrate the state from your local machine to Terraform Cloud.

https://learn.hashicorp.com/terraform/tfc/tfc_migration#migrate-the-state-file

243. How do you configure remote backend with the terraform cloud?

You need to configure in the terraform block

```
terraform {
  backend "remote" {
    hostname      = "app.terraform.io"
    organization  = "<YOUR-ORG-NAME>"

    workspaces {
      name = "state-migration"
    }
  }
}
```

244. What is Run Triggers?

Terraform Cloud's run triggers allow you to link workspaces so that a successful apply in a source workspace will queue a run in the workspace linked to it with a run trigger.

For example, adding new subnets to your network configuration could trigger an update to your application configuration to rebalance servers across the new subnets.

245. What is the benefit of Run Triggers?

When managing complex infrastructure with Terraform Cloud, organizing your configuration into different workspaces helps you to better manage and design your infrastructure.

Configuring run triggers between workspaces allows you to set up infrastructure pipelines as part of your overall deployment strategy.

246. What are the available permissions that terraform clouds can have?

Terraform Cloud teams can have read, plan, write, or admin permissions on individual workspaces.

247. Who can grant permissions on the workspaces?

Organization owners grant permissions by grouping users into teams and giving those teams priviliges based on their need for access to individual workspaces.

248. Which plan do you need to manage teams on Terraform cloud?

Team Plan

249. How can you add users to an organization?

You can add users to an organization by inviting them using their email address.

Even if your team member has not signed up for Terraform Cloud yet, they can still accept the invitation and create a new account.

250. The Terraform Cloud Team plan charges you on a per-user basis. Is this true?

Yes. The Terraform Cloud Team plan is charged on a per-user basis so adding new users to your organization incurs cost.

Conclusion

The Terraform associate exam is multiple-choice, multiple answers, text-based, exam. These sample questions definitely help you prepare for the certification. I would recommend you go through the documentation first and then refer to this afterward or right before the exam.

Terraform Certification DevOps Cloud Computing Software Development