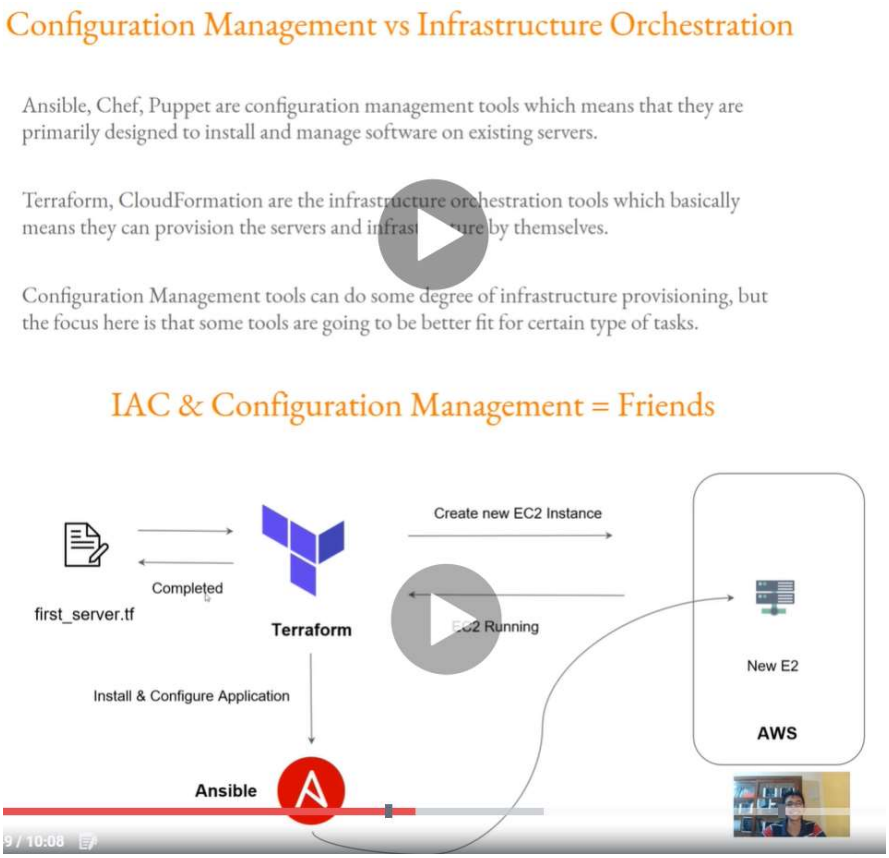
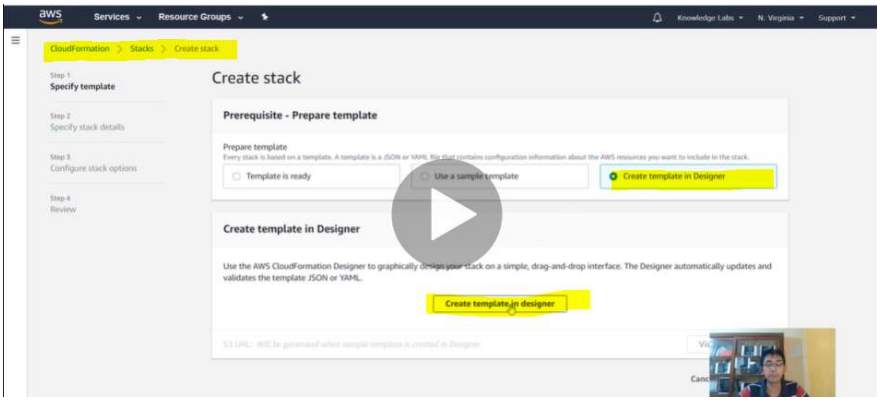
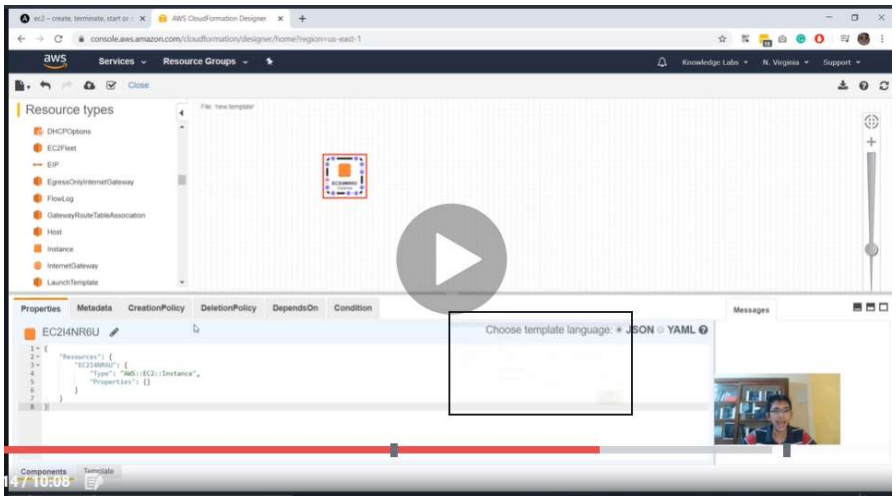
Terraform Notes & Practice



AWS Cloud formation



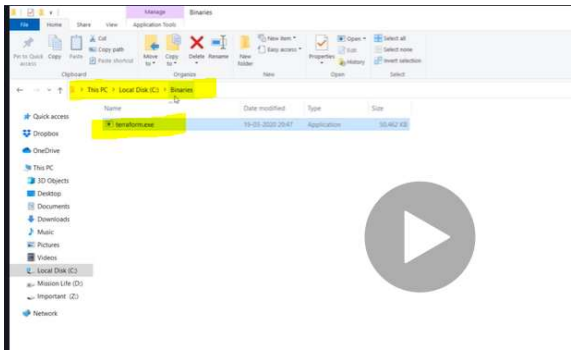


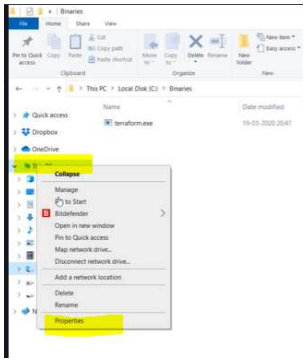
Wednesday, August 12, 2020
5:14 PM

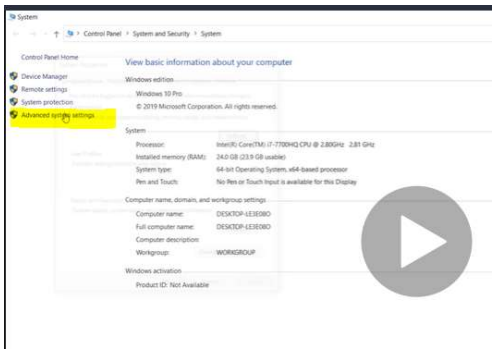
```
hari:~ kammana$ cd Downloads/
hari:Downloads kammana$ mv terraform /usr/local/bin/
hari:Downloads kammana$ terraform --version
Terraform v0.12.4
hari:Downloads kammana$
```

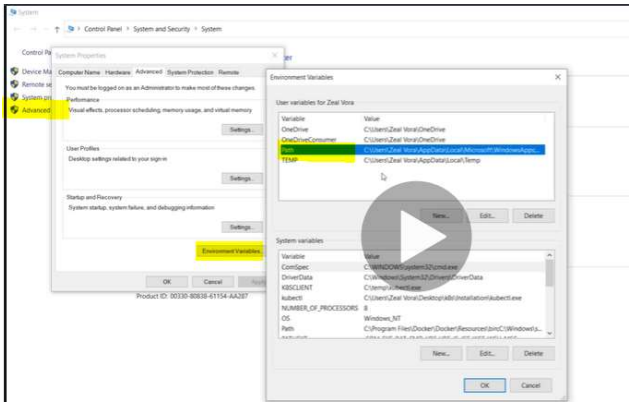
Set the environment variables for Terreform to use it from any directory

Create folder in C drive called Binaries and paste the Terraform.exe in the folder



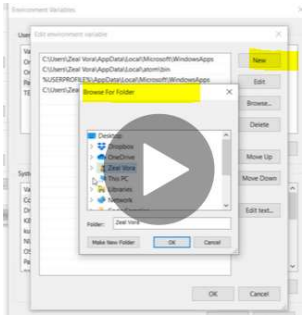






Double click the path and click new path and browse to the terraform.exe file and click ok

Now you can access terraform from any folder in Windows



Linux installation procedure

```
root@281b404f39a6:/# wget https://releases.hashicorp.com/terraform/0.12.24/terraform_0.12.24_linux_amd64.zip
--2020-04-30 20:24:27--  https://releases.hashicorp.com/terraform/0.12.24/terraform_0.12.24_linux_amd64.zip
Resolving releases.hashicorp.com (releases.hashicorp.com)... 151.101.157.183, 2a04:4e42:25::439
Connecting to releases.hashicorp.com (releases.hashicorp.com)|151.101.157.183|:443... connected.
HTTP request sent, awaiting response... 200 OK
Length: 16522081 (16M) [application/zip]
Saving to: 'terraform_0.12.24_linux_amd64.zip'

terraform_0.12.24_linux_amd64 100%[===================================================>]  15.76M  7.58MB/s    in 2.1s

2020-04-30 20:24:29 (7.58 MB/s) - 'terraform_0.12.24_linux_amd64.zip' saved [16522081/16522081]
```

```
root@281b404f39a6:~/terraform# ls
terraform_0.12.24_linux_amd64.zip
root@281b404f39a6:~/terraform# unzip terraform_0.12.24_linux_amd64.zip
Archive:  terraform_0.12.24_linux_amd64.zip
  inflating: terraform
root@281b404f39a6:~/terraform#
```

```
root@281b404f39a6:~/terraform# ls
terraform  terraform_0.12.24_linux_amd64.zip
root@281b404f39a6:~/terraform#
```

```
root@281b404f39a6:~/terraform# ./terraf
```

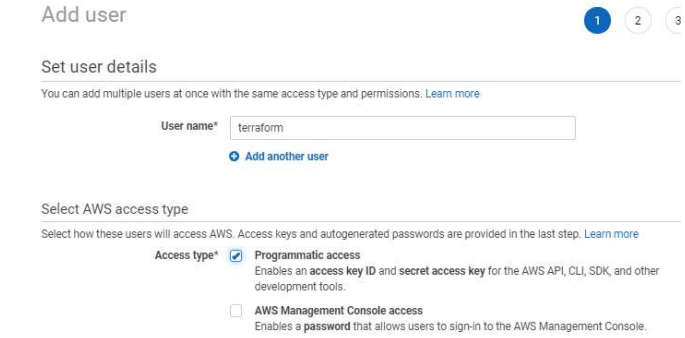
Set the environment variable

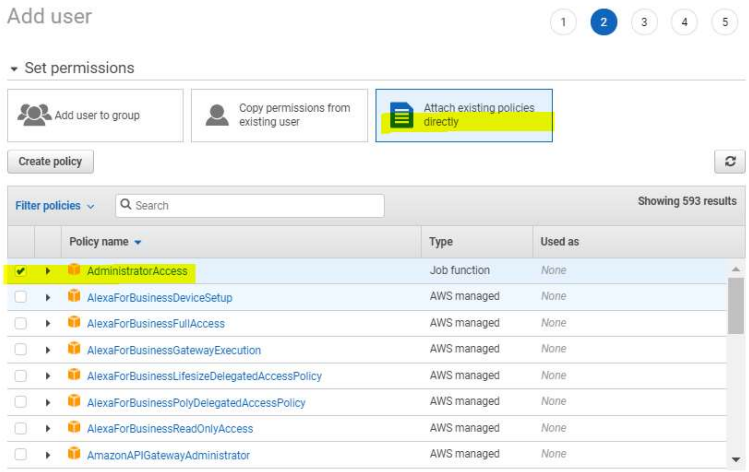
```
root@281b404f39a6:~/terraform# echo $PATH
/root/bin:/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin
root@281b404f39a6:~/terraform# mv terraform /usr/local/bin
root@281b404f39a6:~/terraform#
```

Download the text editor http://atom.io/ for writing the Terraform code

Download visual basic code and install below packages for Terraform

Create Terraform user with Admin right in AWS





User name -terraform

Access key
AKIAQJES3T6VOGOLZ3FF

Secret access key
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

Fist syntax for terraform authentication

Usage:
```
provider "aws" {
  region     = "us-west-2"
  access_key = "my-access-key"
  secret_key = "my-secret-key"
}
```

From <https://registry.terraform.io/providers/hashicorp/aws/latest/docs>

Check for aws resource type

```
# Create a new instance of the latest Ubuntu 20.04 on an
# t3.micro node with an AWS Tag naming it "HelloWorld"
provider "aws" {
  region = "us-west-2"
}
data "aws_ami" "ubuntu" {
  most_recent = true
filter {
    name   = "name"
    values = ["ubuntu/images/hvm-ssd/ubuntu-focal-20.04-amd64-server-*"]
  }
filter {
    name   = "virtualization-type"
    values = ["hvm"]
  }
owners = ["099720109477"] # Canonical
}
resource "aws_instance" "web" {
  ami           = data.aws_ami.ubuntu.id
  instance_type = "t3.micro"
tags = {
    Name = "HelloWorld"
  }
}
```

From <https://registry.terraform.io/providers/hashicorp/aws/latest/docs/resources/instance>

When creating EC2 instance which are required parameters need to be mentioned in the code

Argument Reference

The following arguments are supported:

- `ami` - (Required) The AMI to use for the instance.
- `availability_zone` - (Optional) The AZ to start the instance in.
- `placement_group` - (Optional) The Placement Group to start the instance in.

When code is completed in Visual basic code then get in the folder and enter below **command**

```
provider "aws" {
  region     = "us-east-1"
  access_key = "AKIAQJES3T6VOGOLZ3FF"
  secret_key = "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"
}
resource "aws_instance" "myec2" {
    ami = "ami-0947d2ba12ee1ff75"
    instance_type = "t2.micro"
```

ec2.tf

```
ec2.tf
1  provider "aws" {
2    region     = "us-east-1"
3    access_key = "AKIAQJES3T6VOGOLZ3FF"
4    secret_key = "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"
5  }
6
7  resource "aws_instance" "myec2" {
8      ami = "ami-0947d2ba12ee1ff75"
9      instance_type = "t2.micro"
10 }
```

terraform init
terraform plan
terraform apply

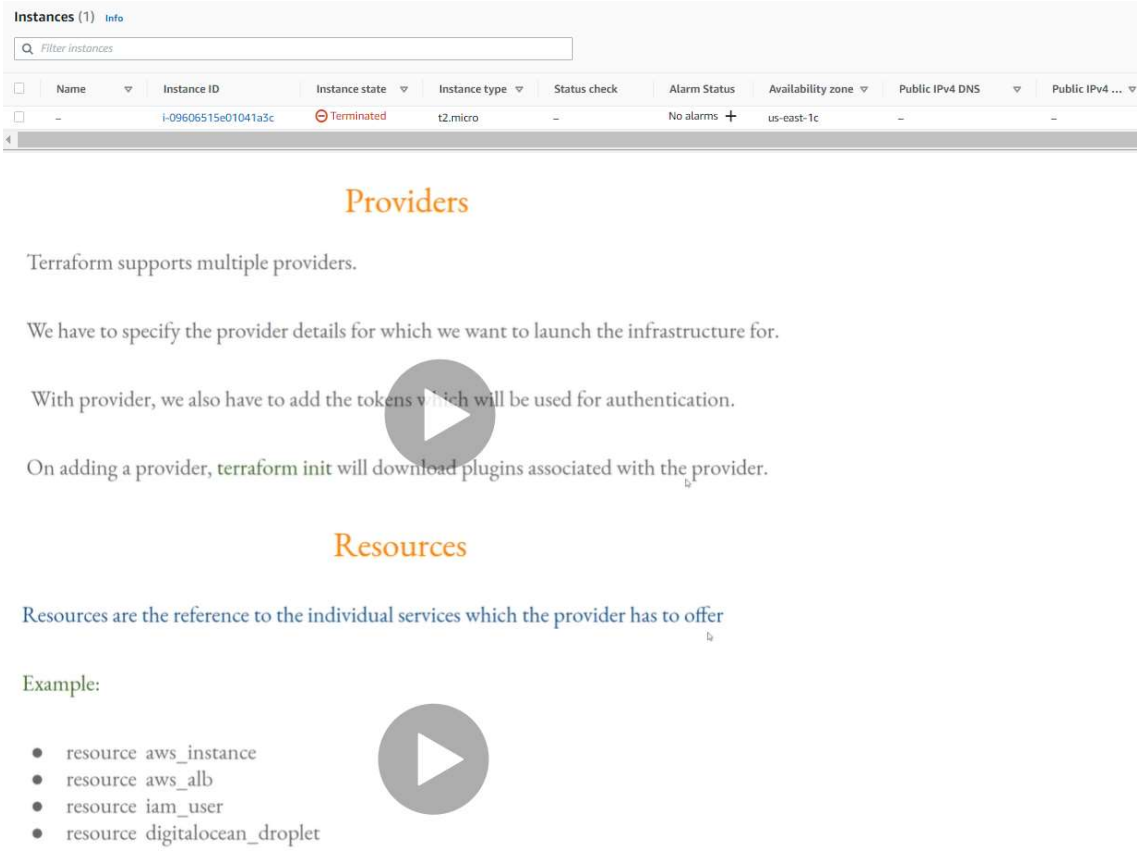
```
C:\Users\nkjaiswal\Desktop\Terraform>terraform init

Initializing the backend...

Initializing provider plugins...
- Using previously-installed hashicorp/aws v3.9.0

The following providers do not have any version constraints in configuration,
so the latest version was installed.

To prevent automatic upgrades to new major versions that may contain breaking
changes, we recommend adding version constraints in a required_providers block
in your configuration, with the constraint strings suggested below.

* hashicorp/aws: version = "~> 3.9.0"

Terraform has been successfully initialized!

You may now begin working with Terraform. Try running "terraform plan" to see
any changes that are required for your infrastructure. All Terraform commands
should now work.

If you ever set or change modules or backend configuration for Terraform,
rerun this command to reinitialize your working directory. If you forget, other
commands will detect it and remind you to do so if necessary.
```

Check which options that was selected like ami,instance type that was selected is listed

```
C:\Users\nkjaiswal\Desktop\Terraform>terraform plan
Refreshing Terraform state in-memory prior to plan...
The refreshed state will be used to calculate this plan, but will not be
persisted to local or remote state storage.


------------------------------------------------------------------------

An execution plan has been generated and is shown below.
Resource actions are indicated with the following symbols:
  + create

Terraform will perform the following actions:

  # aws_instance.myec2 will be created
  + resource "aws_instance" "myec2" {
      + ami                          = "ami-03657b56516ab7912"
      + arn                          = (known after apply)
      + associate_public_ip_address  = (known after apply)
      + availability_zone            = (known after apply)
      + cpu_core_count               = (known after apply)
      + cpu_threads_per_core         = (known after apply)
      + get_password_data            = false
      + host_id                      = (known after apply)
      + id                           = (known after apply)
      + instance_state               = (known after apply)
      + instance_type                = "t2.micro"
      + ipv6_address_count           = (known after apply)
      + ipv6_addresses               = (known after apply)
      + key_name                     = (known after apply)
      + outpost_arn                  = (known after apply)
      + password_data                = (known after apply)
      + placement_group              = (known after apply)
      + primary_network_interface_id = (known after apply)
      + private_dns                  = (known after apply)
      + private_ip                   = (known after apply)
      + public_dns                   = (known after apply)
      + public_ip                    = (known after apply)
      + secondary_private_ips        = (known after apply)
      + security_groups              = (known after apply)
      + source_dest_check            = true
      + subnet_id                    = (known after apply)
```

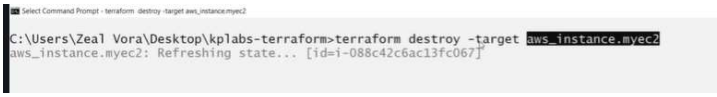
```
C:\Users\nkjaiswal\Desktop\Terraform>terraform apply

An execution plan has been generated and is shown below.
Resource actions are indicated with the following symbols:
  + create

Terraform will perform the following actions:

  # aws_instance.myec2 will be created
  + resource "aws_instance" "myec2" {
      + ami                          = "ami-03657b56516ab7912"
      + arn                          = (known after apply)
      + associate_public_ip_address  = (known after apply)
      + availability_zone            = (known after apply)
      + cpu_core_count               = (known after apply)
      + cpu_threads_per_core         = (known after apply)
      + get_password_data            = false
      + host_id                      = (known after apply)
      + id                           = (known after apply)
      + instance_state               = (known after apply)
      + instance_type                = "t2.micro"
      + ipv6_address_count           = (known after apply)
      + ipv6_addresses               = (known after apply)
      + key_name                     = (known after apply)
      + outpost_arn                  = (known after apply)
      + password_data                = (known after apply)
      + placement_group              = (known after apply)
      + primary_network_interface_id = (known after apply)
      + private_dns                  = (known after apply)
      + private_ip                   = (known after apply)
      + public_dns                   = (known after apply)
      + public_ip                    = (known after apply)
```

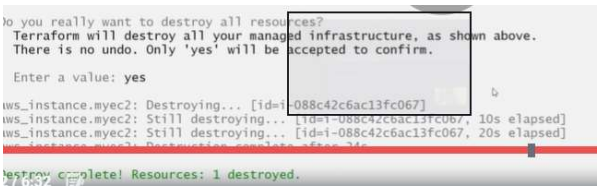
```
Plan: 1 to add, 0 to change, 0 to destroy.

Do you want to perform these actions?
  Terraform will perform the actions described above.
  Only 'yes' will be accepted to approve.

  Enter a value: yes

aws_instance.myec2: Creating...
aws_instance.myec2: Still creating... [10s elapsed]
aws_instance.myec2: Still creating... [20s elapsed]
aws_instance.myec2: Still creating... [30s elapsed]
aws_instance.myec2: Creation complete after 34s [id=i-09606515e01041a3c]

Apply complete! Resources: 1 added, 0 changed, 0 destroyed.
```



Delete EC2 instance by Terraform

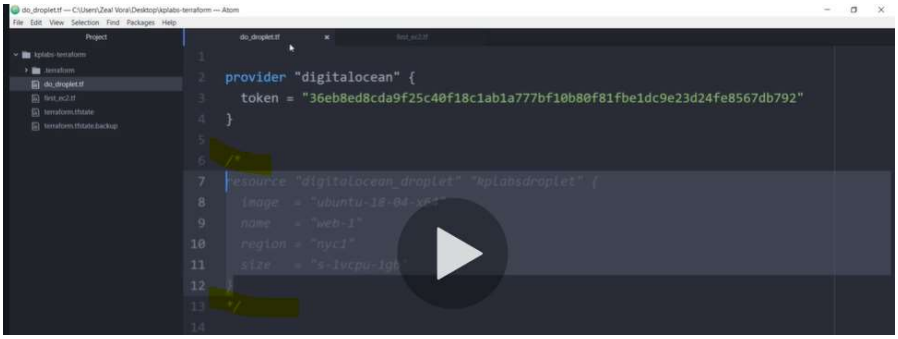
terraform destroy -target aws_instance.myec2





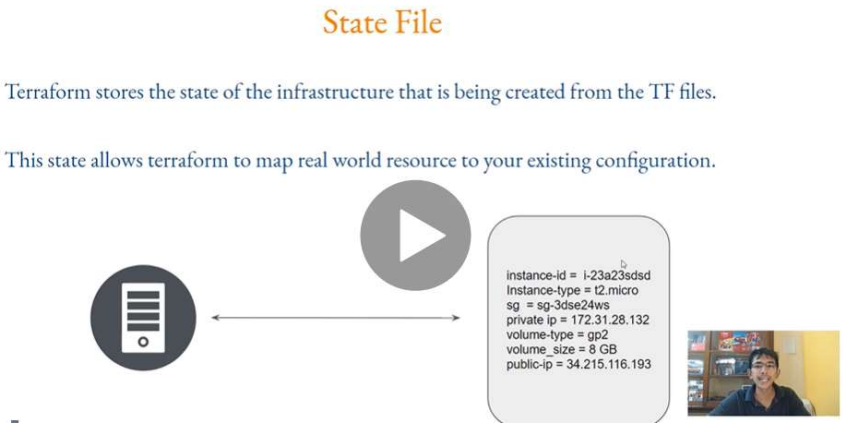
If the resource is commented it will be destroyed

terraform destroy

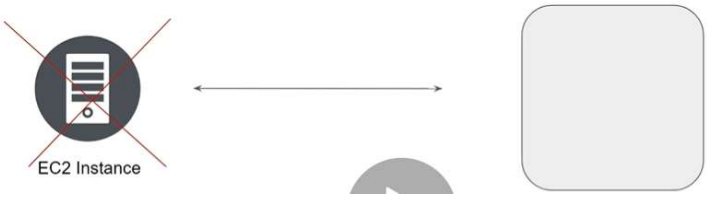


```
C:\Users\Zeal Vora\Desktop\kplabs-terraform>terraform apply
digitalocean_droplet.kplabsdroplet: Refreshing state... [id=152454306]

An execution plan has been generated and is shown below.
Resource actions are indicated with the following symbols:
  - destroy

Terraform will perform the following actions:

  # digitalocean_droplet.kplabsdroplet will be destroyed
  - resource "digitalocean_droplet" "kplabsdroplet" {
      - backups            = false -> null
      - disk               = 25 -> null
      - id                 = "152454306" -> null
```

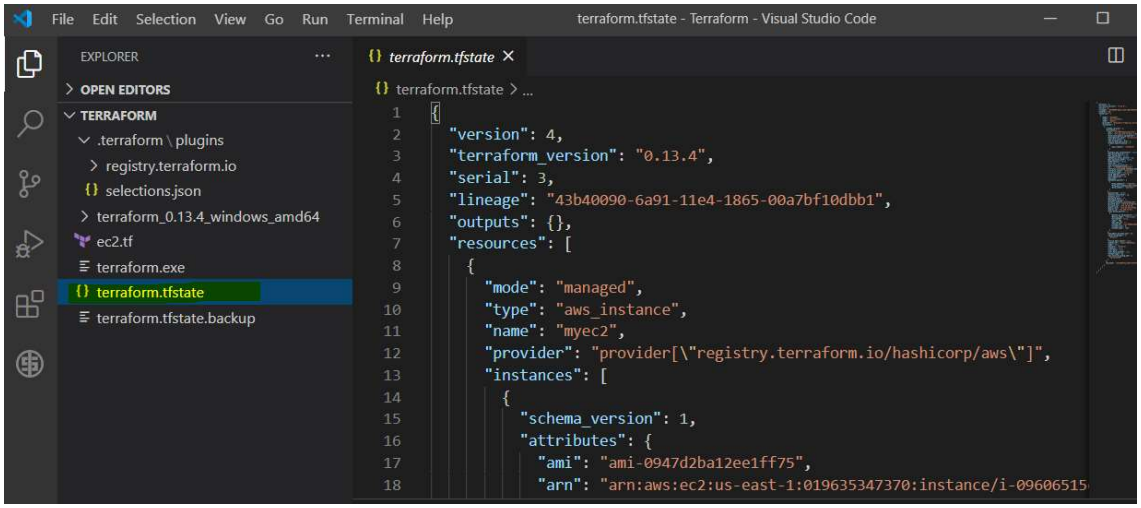
Terraform stores what it has created in its database



Once resource is delete Terraform will also remove it from it database entry



All the information about EC2 instance that was created by Terraform is stored in below Terraform file**(terraform.tfstate)** in the text editor



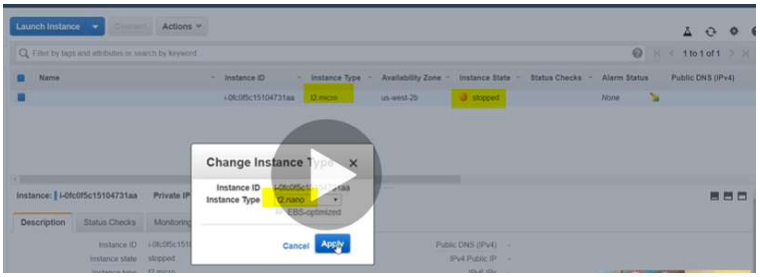
If the EC2 instance is created and if terraform apply is executed for same resourse it will give message

```
C:\Users\Zeal Vora\Desktop\kplabs-terraform>terraform plan
Refreshing Terraform state in-memory prior to plan...
The refreshed state will be used to calculate this plan, but will not be
persisted to local or remote state storage.

aws_instance.myec2: Refreshing state... [id=i-0fc0f5c15104731aa]

------------------------------------------------------------------------

No changes. Infrastructure is up-to-date.

This means that Terraform did not detect any differences between your
configuration and real physical resources that exist. As a result, no
actions need to be performed.
```

Desired state is what is mentioned in the tf file and current state is which is running in production

```
# Desired State and Current State

i) Desired State: EC2 = instance_type = "t2.micro"

ii) Current State:
```

If the instance is stopped and instance type is changed from T2 micro to nano which is current state
Which is visible in **terraform.tfstate file**



If terraform plan is executed then it detected it can see one change and point out what is being changed

```
# Desired State and Current State

i) Desired State: EC2 = instance_type = "t2.micro"

ii) Current State:  "instance_state": "stopped", && "instance_type": "t2.nano",

Desired State == Current State
```

```
C:\Users\Zeal Vora\Desktop\kplabs-terraform>terraform plan
```

```
    }

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

```
Terraform will perform the following actions:

  # aws_instance.myec2 will be updated in-place
  ~ resource "aws_instance" "myec2" {
        ami                          = "ami-082b5a644766e0e6f"
        arn                          = "arn:aws:ec2:us-west-2:182276550051:
        associate_public_ip_address  = false
        availability_zone            = "us-west-2b"
        cpu_core_count               = 1
        cpu_threads_per_core         = 1
        disable_api_termination      = false
        ebs_optimized                = false
        get_password_data            = false
        id                           = "i-0fc0f5c15104731a
        instance_state               = "stopped"
      ~ instance_type                = "t2.nano" -> "t2.mi
        ipv6_address_count           = 0
        ipv6_addresses               = []
        monitoring                   = false
        primary_network_interface_id = "eni-01fc15956fb975475"
        private_dns                  = "ip-172-31-20-18.us-west-2.compute.i
        private_ip                   = "172.31.20.18"
```

From terraform.tf file for creating EC2 instance it was mentioned as t2.micro so it change it to same after u apply
It is showing it is going to change 1 item

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

Recheck this part which something to going to destroy from production

If you change the security group from AWS console but terraform wont detect any changes since it is not part of the tf file.

Here in the terraform tf file security group is not mentioned for EC2 instance therefore the security is changed manually then terraform will not revert back the changes as it is not mentioned in the script.

```
ec2.tf
1  provider "aws" {
2    region     = "us-east-1"
3    access_key = "AKIAQJES3T6VOGOLZ3FF"
4    secret_key = "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"
5  }
6
7  resource "aws_instance" "myec2" {
8      ami = "ami-0947d2ba12ee1ff75"
9      instance_type = "t2.micro"
10 }
```

terraform apply wont detect that changes made in security group becuz it is not mentioned in the tf file

```
C:\Users\Zeal Vora\Desktop\kplabs-terraform>terraform plan
Refreshing Terraform state in-memory prior to plan...
The refreshed state will be used to calculate this plan, but will not be
persisted to local or remote state storage.

aws_instance.myec2: Refreshing state... [id=i-0fc0f5c15104731aa]

------------------------------------------------------------------------

No changes. Infrastructure is up-to-date.

This means that Terraform did not detect any differences between your
configuration and real physical resources that exist. As a result, no
actions need to be performed.
```

So it is advisable to make changes in the tf file to bring in desired changes

To refresh the current state

terraform refresh

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

This 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.

To see the contents of your tfstate file

terraform show

Suppose with TF file below recourses are created

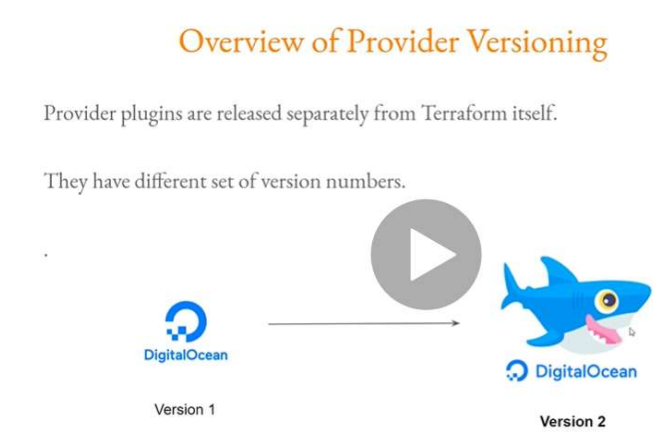
```
root@master:/home/spovedd/Terraform/simple-terraform# terraform state list
aws_instance.web
aws_s3_bucket.b
aws_security_group.sgweb
aws_subnet.public-subnet
aws_vpc.default
```

Someone from the team deleted S3 bucket manually

When terraform refresh is executed it detects manual changes and update the tf file.

When terraform state list is executed then s3 bucket is not there

```
root@master:/home/spovedd/Terraform/simple-terraform# terraform refresh
aws_vpc.default: Refreshing state... [id=vpc-088d8b2c7ef11e2cc]
aws_s3_bucket.b: Refreshing state... [id=my-tf-test-bucket-fhgfgg]
aws_subnet.public-subnet: Refreshing state... [id=subnet-0dfd0bdbb37d26d33]
aws_security_group.sgweb: Refreshing state... [id=sg-09f1ac2794e829cca]
aws_instance.web: Refreshing state... [id=i-05d61442b4e4beff6]
root@master:/home/spovedd/Terraform/simple-terraform# terraform state list
aws_instance.web
aws_security_group.sgweb
aws_subnet.public-subnet
aws_vpc.default
```



Explicitly Setting Provider Version

During terraform init, if version argument is not specified, the most recent provider will be downloaded during initialization.

For production use, you should constrain the acceptable provider versions via configuration, to ensure that new versions with breaking changes will not be automatically installed.

```
provider "aws" {
  region     = "us-west-2"
  version    = "2.7"
}
```

If no version is mentioned in the tf file then during init phase it will download the latest version which might break your existing infrastructure
So it is important to mention the version in tf file

```
provider-versioning.tf
provider "aws" {
  region     = "us-west-2"
  access_key = "AKIAQIW66DN2W7WOYRGY"
  secret_key = "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"
}

resource "aws_instance" "myec2" {
   ami = "ami-082b5a644766e0e6f"
   instance_type = "t2.micro"
}
```

```
Command Prompt - terraform init
C:\Users\Zeal Vora\Desktop\terraform>terraform init

Initializing the backend...

Initializing provider plugins...
- Checking for available provider plugins...
- Downloading plugin for provider "aws" (hashicorp/aws) 2.58.0...
```

```
Select Command Prompt
C:\Users\Zeal Vora\Desktop\terraform>terraform init

Initializing the backend...

Initializing provider plugins...
- Checking for available provider plugins...
- Downloading plugin for provider "aws" (hashicorp/aws) 2.58.0...

The following providers do not have any version constraints in configuration,
so the latest version was installed.

To prevent automatic upgrades to new major versions that may contain breaking
changes, it is recommended to add version = "..." constraints to the
corresponding provider blocks in configuration, with the constraint strings
suggested below.

* provider.aws: version = "~> 2.58"

Terraform has been successfully initialized!
```

If version is mentioned in the tf file them it will download plugin to same version

```
provider-versioning.tf — C:\Users\Zeal Vora\Desktop\terraform — Atom
File  Edit  View  Selection  Find  Packages  Help
provider-versioning.tf
provider "aws" {
  region     = "us-west-2"
  access_key = "AKIAQIW66DN2W7WOYRGY"
  secret_key = "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"
  version    = "2.7"
}

resource "aws_instance" "myec2" {
   ami = "ami-082b5a644766e0e6f"
   instance_type = "t2.micro"
}
```

```
Select Command Prompt - terraform init

C:\Users\Zeal Vora\Desktop\terraform>terraform init

Initializing the backend...

Initializing provider plugins...
- Checking for available provider plugins...
- Downloading plugin for provider "aws" (hashicorp/aws) 2.7.0...
```

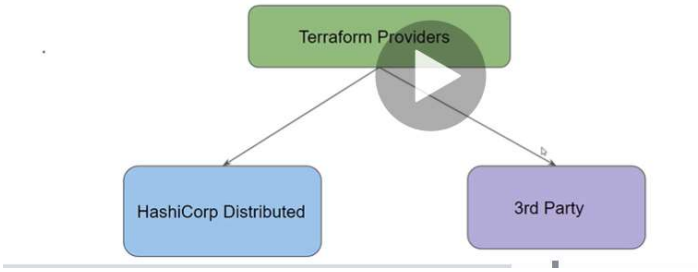
Arguments for Specifying provider

There are multiple ways for specifying the version of a provider.

Version Number Arguments	Description
>=1.0	Greater than equal to the version
<=1.0	Less than equal to the version
~>2.0	Any version in the 2.X range.
>=2.10,<=2.30	Any version between 2.10 and 2.30

Terraform Provider Types

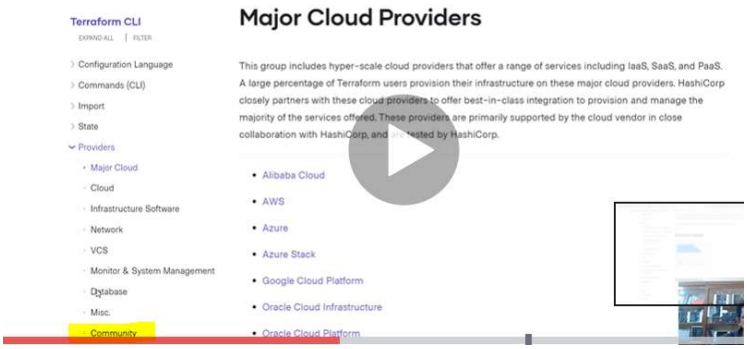
There are two major categories for terraform providers.



HashiCorp Distributed providers can be downloaded automatically during terraform init.

terraform init cannot automatically download providers that are not distributed by HashiCorp

3rd party is referred to Community section where plugins needs to be manually installed provided by vendor



Terraform does not have wavefront provider plugin and gives below message

```
Command Prompt - docker exec -it terraform bash
  GNU nano 2.9.8

provider "wavefront" {
  address = "spaceape.wavefront.com"
}
```

```
bash-4.2# terraform init

Initializing the backend...

Initializing provider plugins...
- Checking for available provider plugins...

Provider "wavefront" not available for installation.

A provider named "wavefront" could not be found in the Terraform Registry.

This may result from mistyping the provider name, or the given provider may
be a third-party provider that cannot be installed automatically.

In the latter case, the plugin must be installed manually by locating and
downloading a suitable distribution package and placing the plugin's executable
file in the following directory:
    terraform.d/plugins/linux_amd64

Terraform detects necessary plugins by inspecting the configuration and state.
To view the provider versions requested by each module, run
"terraform providers".


Error: no provider exists with the given name
```

Path to install the plugin for third party providers

Configuring 3rd Party Provider

Third-party providers must be manually installed, since terraform init cannot autom download them.

Install third-party providers by placing their plugin executables in the user plugins di

Operating system	User plugins directory
Windows	%APPDATA%\terraform.d\plugins
All other systems	~/.terraform.d/plugins

```
bash-4.2# wget https://github.com/spaceapegames/terraform-provider-wavefront/releases/download/v2.1.1/terraform-provid
er-wavefront_v2.1.1_darwin_amd64
```

```
bash-4.2# ls
aws.tf  terraform-provider-wavefront_v2.1.1_darwin_amd64  wavefront.tf
bash-4.2# mv terraform-provider-wavefront_v2.1.1_darwin_amd64 terraform-provider-wavefront_v2.1.1
bash-4.2# ls
aws.tf  terraform-provider-wavefront_v2.1.1  wavefront.tf
bash-4.2#
```

```
bash-4.2# mkdir -p ~/.terraform.d/plugins
bash-4.2# mv terraform-provider-wavefront_v2.1.1 ~/.terraform.d/plugins/
bash-4.2#
```

```
bash-4.2# terraform init

Initializing the backend...

Initializing provider plugins...

The following providers do not have any version constraints in configuration,
so the latest version was installed.

To prevent automatic upgrades to new major versions that may contain breaking
changes, it is recommended to add version = "..." constraints to the
corresponding provider blocks in configuration, with the constraint strings
suggested below.

* provider.wavefront: version = "~> 2.1"

Terraform has been successfully initialized!
```

Use different folders for every deployment

Overview of the Format

We tend to use a different folder for each practical that we do in the course.

This allows us to be more systematic and allows easier revisit in-case required.

Lecture Name	Folder Names
Create First EC2 Instance	folder1
Tainting resource	folder2
Conditional Expression	folder3

Destroy Resource After Practical

We know how to destroy resources by now

terraform destroy

After you have completed your practical, make sure you destroy the resource before moving to the next practical.

This is easier if you are maintaining separate folder for each practical.

Understanding Attributes

Terraform has capability to output the attribute of a resource with the output values.

Example:

ec2_public_ip = 35.161.21.197
bucket_identifier = terraform-test-kplabs.s3.amazonaws.com

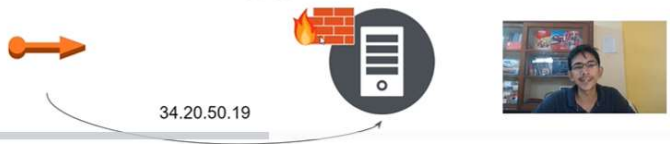
EIP is elastic IP

Attributes are important

An outputed attributes can not only be used for the user reference but it can also act as a input to other resources being created via terraform

Let's understand this with an example:

After EIP gets created, it's IP address should automatically get whitelisted in the security group.



https://github.com/zealvora/terraform-beginner-to-advanced-resource

First we are creating elastic ip and s3 bucket

```
att.tf
1  provider "aws" {
2    region     = "us-west-2"
3    access_key = "AKIAQJES3T6VOGOLZ3FF"
4    secret_key = "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"
5  }
6
7  resource "aws_eip" "lb" {
8    vpc      = true
9  }
10
11 /*
12   output "eip" {
13     value = aws_eip.lb
14   }
15 */
16 resource "aws_s3_bucket" "mys3" {
17   bucket = "kplabs-attribute-demo-001"
18 }
```

To comment

First terraform plan -check what is will be created

```
C:\Users\nkjaiswal\Desktop\Terraform>terraform plan
Refreshing Terraform state in-memory prior to plan...
The refreshed state will be used to calculate this plan, but will not be
persisted to local or remote state storage.

aws_instance.myec2: Refreshing state... [id=i-09606515e01041a3c]

------------------------------------------------------------------------

An execution plan has been generated and is shown below.
Resource actions are indicated with the following symbols:
  + create

Terraform will perform the following actions:

  # aws_eip.lb will be created
  + resource "aws_eip" "lb" {
      + allocation_id     = (known after apply)
      + association_id    = (known after apply)
      + customer_owned_ip = (known after apply)
      + domain            = (known after apply)
      + id                = (known after apply)
      + instance          = (known after apply)
      + network_interface = (known after apply)
      + private_dns       = (known after apply)
      + private_ip        = (known after apply)
      + public_dns        = (known after apply)
      + public_ip         = (known after apply)
      + public_ipv4_pool  = (known after apply)
      + vpc               = true
    }

  # aws_s3_bucket.mys3 will be created
  + resource "aws_s3_bucket" "mys3" {
      + acceleration_status         = (known after apply)
      + acl                         = "private"
      + arn                         = (known after apply)
      + bucket                      = "kplabs-attribute-demo-001"
      + bucket_domain_name          = (known after apply)
      + bucket_regional_domain_name = (known after apply)
      + force_destroy               = false
      + hosted_zone_id              = (known after apply)
      + id                          = (known after apply)
      + region                      = (known after apply)
      + request_payer               = (known after apply)
      + website_domain              = (known after apply)
      + website_endpoint            = (known after apply)

      + versioning {
```

Terraform destroy

```
Terraform will perform the following actions:

  # aws_eip.lb will be destroyed
  - resource "aws_eip" "lb" {
      - domain           = "vpc" -> null
      - id               = "eipalloc-09b7be6c009a8e202" -> null
      - public_dns       = "ec2-54-71-66-108.us-west-2.compute.amazonaws.com" -> null
      - public_ip        = "54.71.66.108" -> null
      - public_ipv4_pool = "amazon" -> null
      - tags             = {} -> null
      - vpc              = true -> null
    }

  # aws_s3_bucket.mys3 will be destroyed
  - resource "aws_s3_bucket" "mys3" {
      - acl                         = "private" -> null
      - arn                         = "arn:aws:s3:::kp0009labs-attribute-demo-001" -> null
      - bucket                      = "kp0009labs-attribute-demo-001" -> null
      - bucket_domain_name          = "kp0009labs-attribute-demo-001.s3.amazonaws.com" -> null
      - bucket_regional_domain_name = "kp0009labs-attribute-demo-001.s3.us-west-2.amazonaws.com" -> null
      - force_destroy               = false -> null
      - hosted_zone_id              = "Z3BJ6K6RIION7M" -> null
      - id                          = "kp0009labs-attribute-demo-001" -> null
      - region                      = "us-west-2" -> null
      - request_payer               = "BucketOwner" -> null
      - tags                        = {} -> null

      - versioning {
          - enabled    = false -> null
          - mfa_delete = false -> null
        }
    }

Plan: 0 to add, 0 to change, 2 to 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

aws_eip.lb: Destroying... [id=eipalloc-09b7be6c009a8e202]
aws_s3_bucket.mys3: Destroying... [id=kp0009labs-attribute-demo-001]
aws_s3_bucket.mys3: Destruction complete after 1s
aws_eip.lb: Destruction complete after 2s
```

```
Plan: 0 to add, 0 to change, 2 to 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

aws_eip.lb: Destroying... [id=eipalloc-09b7be6c009a8e202]
aws_s3_bucket.mys3: Destroying... [id=kp0009labs-attribute-demo-001]
aws_s3_bucket.mys3: Destruction complete after 1s
aws_eip.lb: Destruction complete after 2s

Destroy complete! Resources: 2 destroyed.
```

After removing comment from ouput and apply the terraform tft it provides details output of s3 bucket and elastic ip

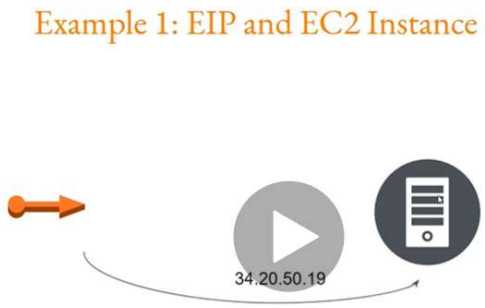
```
  secret_key = "N6iS8bvgQAf3OEmxgckm5Gr0lWNx1TMfH3FoZ9
}

resource "aws_eip" "lb" {
  vpc      = true
}

output "eip" {
  value = aws_eip.lb
}

resource "aws_s3_bucket" "mys3" {
  bucket = "kp0009labs-attribute-demo-001"
}

output "mys3bucket" {
  value = aws_s3_bucket.mys3
}
```

```
aws_eip.lb: Creation complete after 1s [id=eipalloc-07f5dd16558b13eb7]
aws_s3_bucket.mys3: Still creating... [10s elapsed]
aws_s3_bucket.mys3: Creation complete after 10s [id=kp0009labs-attribute-demo-001]

Apply complete! Resources: 2 added, 0 changed, 0 destroyed.

Outputs:

eip = {
  "association_id" = ""
  "customer_owned_ip" = ""
  "customer_owned_ipv4_pool" = ""
  "domain" = "vpc"
  "id" = "eipalloc-07f5dd16558b13eb7"
  "instance" = ""
  "network_interface" = ""
  "private_ip" = ""
  "public_dns" = "ec2-52-11-154-185.us-west-2.compute.amazonaws.com"
  "public_ip" = "52.11.154.185"
  "public_ipv4_pool" = "amazon"
  "vpc" = true
}
mys3bucket = {
  "acceleration_status" = ""
  "acl" = "private"
  "arn" = "arn:aws:s3:::kp0009labs-attribute-demo-001"
  "bucket" = "kp0009labs-attribute-demo-001"
  "bucket_domain_name" = "kp0009labs-attribute-demo-001.s3.amazonaws.com"
  "bucket_regional_domain_name" = "kp0009labs-attribute-demo-001.s3.us-west-2.amazonaws.com"
  "cors_rule" = []
  "force_destroy" = false
  "grant" = []
  "hosted_zone_id" = "Z3BJ6K6RIION7M"
  "id" = "kp0009labs-attribute-demo-001"
  "lifecycle_rule" = []
  "logging" = []
  "object_lock_configuration" = []
  "region" = "us-west-2"
  "replication_configuration" = []
  "request_payer" = "BucketOwner"
  "server_side_encryption_configuration" = []
  "versioning" = [
    {
      "enabled" = false
      "mfa_delete" = false
    },
  ]
  "website" = []
}
```

Attributes of elastic ip address are below

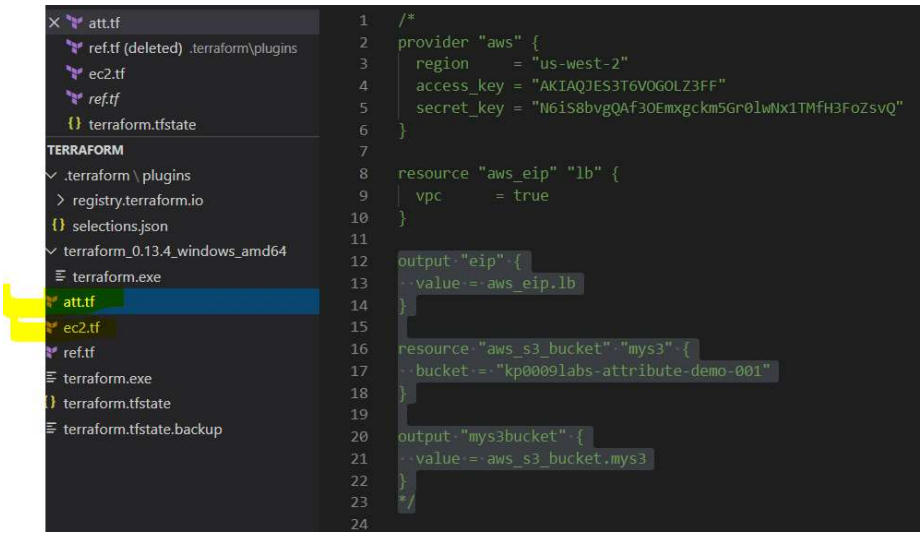
Create EC2 instance and Elastic ip and get it associated to instance

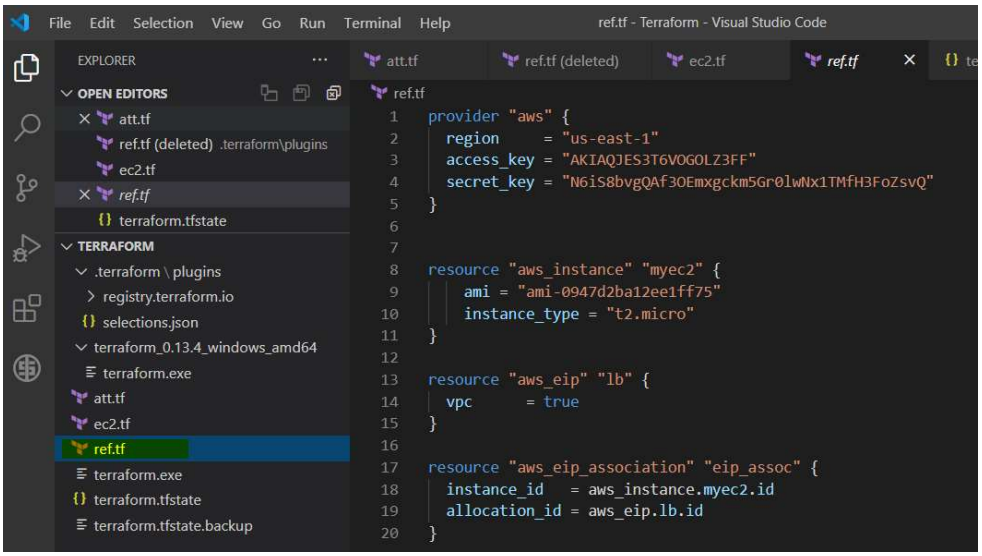
Create Elastic ip and get it associated to security group



Example 2: EIP and Security Group

Commented both the tf files and only ref.tf was created to include creation of ec2 instances and elastic ip





All the tf files should be under .terrform folder otherwise there will be problem

Terraform

File Home Share View

This PC > Desktop > Terraform

Name	Date modified	Type	Size
.terraform	10/8/2020 2:14 PM	File folder	
terraform_0.13.4_windows_amd64	10/8/2020 2:33 PM	File folder	
att.tf	10/9/2020 3:34 PM	TF File	1 KB
ec2.tf	10/9/2020 3:23 PM	TF File	1 KB
ref.tf	10/9/2020 3:13 PM	TF File	1 KB
terraform	10/8/2020 11:12 A...	Application	83,505 KB
terraform.tfstate	10/9/2020 2:21 PM	TFSTATE File	1 KB
terraform.tfstate.backup	10/9/2020 2:21 PM	BACKUP File	13 KB

```
C:\Users\nkjaiswal\Desktop\Terraform>terraform plan
```

```
Plan: 3 to add, 0 to change, 0 to destroy.

------------------------------------------------------------------------

Note: You didn't specify an "-out" parameter to save this plan, so Terraform
can't guarantee that exactly these actions will be performed if
"terraform apply" is subsequently run.
```

Instances (1/1) Info

Filter instances

Name	Instance ID	Instance state	Instance type	Status check	Alarm Status	Availability zone	Public IPv4 DNS	Public IPv4 ...	Elastic Ip	IP
–	i-0f62b329144ca5b65	Running	t2.micro	Initializing	No alarms +	us-east-1d	ec2-54-147-144-147.c...	54.147.144.147	54.147.144.147	–

Instance: i-0f62b329144ca5b65

Details | Security | Networking | Storage | Status Checks | Monitoring | Tags

Instance summary Info

Instance ID
i-0f62b329144ca5b65

Instance state
Running

Instance type
t2.micro

Public IPv4 address
54.147.144.147 | open address

Public IPv4 DNS
ec2-54-147-144-147.compute-1.amazonaws.com | open address

Elastic IP addresses
54.147.144.147 (Public IP)

New EC2 Experience
Tell us what you think

EC2 Dashboard New
Events New
Tags
Limits

Instances
Instances New
Instance Types
Launch Templates
Spot Requests
Savings Plans
Reserved Instances
Dedicated Hosts New
Scheduled Instances
Capacity Reservations

Images
AMIs

Elastic Block Store
Volumes
Snapshots
Lifecycle Manager

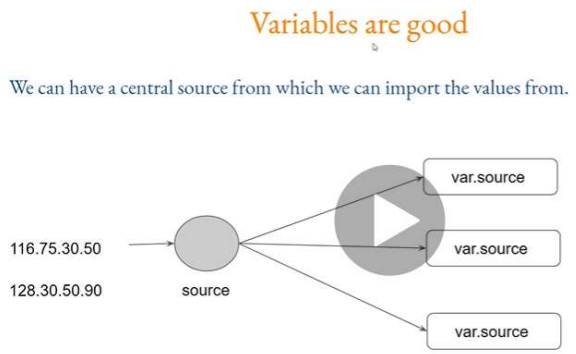
Network & Security
Security Groups New
Elastic IPs New

Elastic IP addresses (1/1)

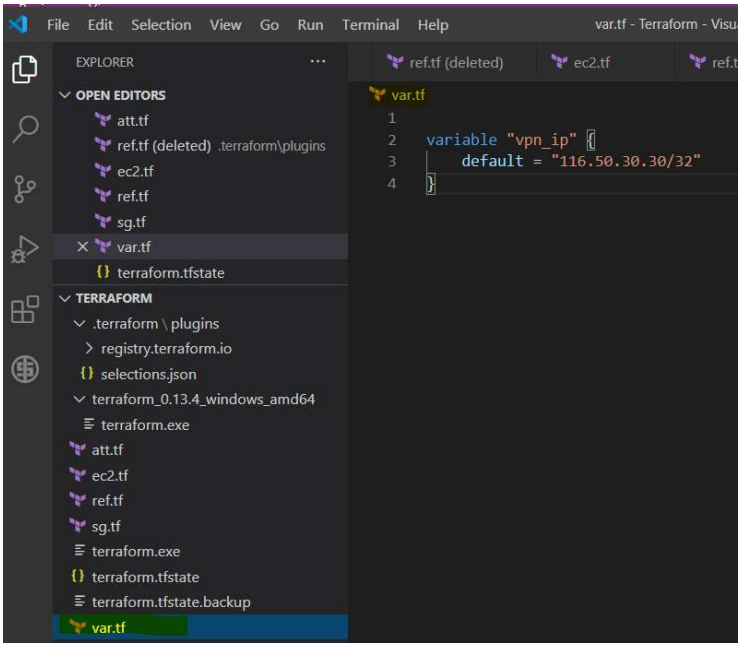
Filter Elastic IP addresses

Name	Allocated IPv4 add...	Type
–	54.147.144.147	Public IP

Variables are be called out wherever they are required to replace with new values in required resource



Created var with ip address



In this CIDR blocks mentioned the variable **(var.vpn_ip)**

File Edit Selection View Go Run Terminal Help sg.tf - Terraform - Visual Studio Code

EXPLORER

OPEN EDITORS
- att.tf
- ref.tf (deleted) .terraform\plugins
- ec2.tf
- ref.tf
- sg.tf
- var.tf
- terraform.tfstate

TERRAFORM
- .terraform \ plugins
  - registry.terraform.io
  - selections.json
  - terraform_0.13.4_windows_amd64
    - terraform.exe
- att.tf
- ec2.tf
- ref.tf
- sg.tf
- terraform.exe
- terraform.tfstate
- terraform.tfstate.backup
- var.tf

att.tf | ref.tf (deleted) | ec2.tf | ref.tf | sg.tf

sg.tf

```
provider "aws" {
  region     = "us-east-1"
  access_key = "AKIAQJES3T6VOGOLZ3FF"
  secret_key = "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"
}


resource "aws_security_group" "allow_tls" {
  name        = "allow_tls"


  ingress {
    from_port   = 443
    to_port     = 443
    protocol    = "tcp"
    cidr_blocks = [var.vpn_ip]
  }

  ingress {
    from_port   = 80
    to_port     = 80
    protocol    = "tcp"
    cidr_blocks = [var.vpn_ip]
  }

  ingress {
    from_port   = 53
    to_port     = 53
    protocol    = "tcp"
    cidr_blocks = [var.vpn_ip]

  }
}
```

Security Groups (1/2) Info

Actions ▼ | Create security

Filter security groups

Name	Security group ID	Security group name	VPC ID	Description	Owner
–	sg-03d02f5ae3541e059	allow_tls	vpc-ac8141d1	Managed by Terraform	019635347370
–	sg-1a727724	default	vpc-ac8141d1	default VPC security gr...	019635347370

Inbound rules | Edit inbound rules

Type	Protocol	Port range	Source	Description - optional
HTTP	TCP	80	116.50.50.30/32	-
DNS (TCP)	TCP	53	116.50.50.30/32	-
HTTPS	TCP	443	116.50.50.30/32	-

Then change ip for all inbound rules by adding new ips in the variable instead of manually editing for all parameters

Inbound rules | Edit inbound rules

Type	Protocol	Port range	Source	Description - optional
HTTP	TCP	80	116.50.50.30/32	-
DNS (TCP)	TCP	53	116.50.50.30/32	-
HTTPS	TCP	443	116.50.50.30/32	-

Multiple Approaches to Variable Assignment

Variables in Terraform can be assigned values in multiple ways.

Some of these include:

- Environment variables
- Command Line Flags
- From a File
- Variable Defaults

Variable defaults

Created variable and added instance type as t2.micro which is default instance type whenever new instance is created

```
var1.tf
variable "instance_type" {
    default = "t2.micro"
}
```

```
ec2_var.tf

provider "aws" {
  region     = "us-east-1"
  access_key = "AKIAQJES3T6VOGOLZ3FF"
  secret_key = "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"
}

resource "aws_instance" "myec2" {
    ami = "ami-0947d2ba12ee1ff75"
    instance_type = var.instance_type
}

```

After executing terraform apply it showed instance type is t2.micro

```
# aws_instance.myec2 will be created
+ resource "aws_instance" "myec2" {
    + ami                          = "ami-0947d2ba12ee1ff75"
    + arn                          = (known after apply)
    + associate_public_ip_address  = (known after apply)
    + availability_zone            = (known after apply)
    + cpu_core_count               = (known after apply)
    + cpu_threads_per_core         = (known after apply)
    + get_password_data            = false
    + host_id                      = (known after apply)
    + id                           = (known after apply)
    + instance_state               = (known after apply)
    + instance_type                = "t2.micro"
    + ipv6_address_count           = (known after apply)
    + ipv6_addresses               = (known after apply)
```

- From a File

Create terraform.tfvars (should use same name of the file) then it will create instance with size t2.small rather than the default t2.micro
This is used when you don't want to create default t2.micro instances by default.

```
ec2_var.tf
provider "aws" {
  region     = "us-east-1"
  access_key = "AKIAQJES3T6VOGOLZ3FF"
  secret_key = "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"
}

resource "aws_instance" "myec2" {
    ami = "ami-0947d2ba12ee1ff75"
    instance_type = var.instance_type
}

```

```
var1.tf
variable "instance_type" {
    default = "t2.micro"
}

```

```
terraform.tfvars
instance_type="t2.small"

```

terraform plan

Select Command Prompt

```
An execution plan has been generated and is shown below.
Resource actions are indicated with the following symbols:
  + create

Terraform will perform the following actions:

  # aws_instance.myec2 will be created
  + resource "aws_instance" "myec2" {
      + ami                          = "ami-0947d2ba12ee1ff75"
      + arn                          = (known after apply)
      + associate_public_ip_address  = (known after apply)
      + availability_zone            = (known after apply)
      + cpu_core_count               = (known after apply)
      + cpu_threads_per_core         = (known after apply)
      + get_password_data            = false
      + host_id                      = (known after apply)
      + id                           = (known after apply)
      + instance_state               = (known after apply)
      + instance_type                = "t2.small"
      + ipv6_address_count           = (known after apply)
      + ipv6_addresses               = (known after apply)
```

If the terraform.tfvars file has no value then by default t2.micro in created

terraform.tfvars

```
1
2
```

terraform plan

Select Command Prompt

```
Terraform will perform the following actions:

  # aws_instance.myec2 will be created
  + resource "aws_instance" "myec2" {
      + ami                          = "ami-0947d2ba12ee1ff75"
      + arn                          = (known after apply)
      + associate_public_ip_address  = (known after apply)
      + availability_zone            = (known after apply)
      + cpu_core_count               = (known after apply)
      + cpu_threads_per_core         = (known after apply)
      + get_password_data            = false
      + host_id                      = (known after apply)
      + id                           = (known after apply)
      + instance_state               = (known after apply)
      + instance_type                = "t2.micro"
      + ipv6_address_count           = (known after apply)
      + ipv6_addresses               = (known after apply)
```

Overview of Type Constraints

The type argument in a variable block allows you to restrict the type of value that will be accepted as the value for a variable

```
variable "image_id" {
  type = string
}
```

If no type constraint is set then a value of any type is accepted.

Example Use-Case

Every employee in Medium Corp is assigned a Identification Number.

Any resource that employee creates should be created with the name of the identification number only.

variables.tf	terraform.tfvars
variable “instance_name” {}	instance_name=”john-123”

Example Use-Case

Every employee in Medium Corp is assigned a Identification Number.

Any EC2 instance that employee creates should be created using the identification number only.

variables.tf	terraform.tfvars

variable “instance_name” {
type=number
}
	instance_name=”john-123”

Overview of Data Types

Type Keywords	Description
string	Sequence of Unicode characters representing some text, like "hello".
list	Sequential list of values identified by their position. Starts with 0 [“mumbai”, “singapore”, “usa”]
map	a group of values identified by named labels, like {name = "Mabel", age = 52}.
number	Example: 200

var2.tf

```
variable "elb_name" {
    type = string
}

variable "az" {
    type = list
}

variable "timeout" {
    type = number
}
```

er with elb.tft

n variable and its type which is very important to mention

he variable defined

elb.tf

```
provider "aws" {
  region     = "us-east-1"
  access_key = "AKIAQJES3T6VOGOLZ3FF"
  secret_key = "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"
}

# Create a new load balancer
resource "aws_elb" "bar" {
  name               = var.elb_name
  availability_zones = var.az



  listener {
    instance_port     = 8000
    instance_protocol = "http"
    lb_port           = 80
    lb_protocol       = "http"
  }


  health_check {
    healthy_threshold   = 2
    unhealthy_threshold = 2
    timeout             = 3
    target              = "HTTP:8000/"
    interval            = 30
  }
```

ref.tf .\

```
:-1b"]
```

```
C:\Users\nkjaiswal\Desktop\Terraform>terraform plan
Refreshing Terraform state in-memory prior to plan...
The refreshed state will be used to calculate this plan, but will not be
persisted to local or remote state storage.


------------------------------------------------------------------------

An execution plan has been generated and is shown below.
Resource actions are indicated with the following symbols:
  + create
```

```
25    timeout             = 3
26    target              = "HTTP:8000/"
27    interval            = 30
28  }
29
30
31  cross_zone_load_balancing   = true
32  idle_timeout                = var.timeout
33  connection_draining         = true
34  connection_draining_timeout = var.timeout
35
36  tags = {
37    Name = "foobar-terraform-elb"
38  }
39 }
```

```
An execution plan has been generated and is shown below.
Resource actions are indicated with the following symbols:
  + create

Terraform will perform the following actions:
```

```
Terraform will perform the following actions:

  # aws_elb.bar will be created
  + resource "aws_elb" "bar" {
      + arn                         = (known after apply)
      + availability_zones          = [
          + "us-west-1a",
          + "us-west-1b",
        ]
      + connection_draining         = true
      + connection_draining_timeout = 400
      + cross_zone_load_balancing   = true
      + dns_name                    = (known after apply)
      + id                          = (known after apply)
      + idle_timeout                = 400
      + instances                   = (known after apply)
      + internal                    = (known after apply)
      + name                        = "myelb"
      + security_groups             = (known after apply)
```

In variable.tft file

Line #9 t2.micro can be replaced by

```
instance_type = var.types["us-west-2"]
```

```
fetch-values-variables.tf
6
7  resource "aws_instance" "myec2" {
8    ami = "ami-082b5a644766e0e6f"
9    instance_type = "t2.micro"
10 }
11
12 variable "list" {
13   type = list
14   default = ["m5.large","m5.xlarge","t2.medium"]
15 }
16
17 variable "types" {
18   type = map
19   default = {
20     us-east-1 = "t2.micro"
21     us-west-2 = "t2.nano"
22     ap-south-1 = "t2.small"
23   }
```

Here the instance type created is t2.small

```
fetch-values-variables.tf
6
7  resource "aws_instance" "myec2" {
8    ami = "ami-082b5a644766e0e6f"
9    instance_type = var.types["ap-south-1"]
10 }
11
12 variable "list" {
13   type = list
14   default = ["m5.large","m5.xlarge","t2.medium"]
15 }
16
17 variable "types" {
18   type = map
19   default = {
20     us-east-1 = "t2.micro"
21     us-west-2 = "t2.nano"
22     ap-south-1 = "t2.small"
23   }
```

Here the instance type created is **m5.large**

```
+ id                          = (known after a
+ instance_state              = (known after a
+ instance_type               = "m5.large"
+ ipv6_address_count          = (known after a
```

```
fetch-values-variables.tf

resource "aws_instance" "myec2" {
   ami = "ami-082b5a644766e0e6f"
   instance_type = var.list["ap-south-1"]
}

variable "list" {
  type = list
  default = ["m5.large","m5.xlarge","t2.medium"]
}
                 0        1          2

variable "types" {
  type = map
  default = {
    us-east-1 = "t2.micro"
    us-west-2 = "t2.nano"
    ap-south-1 = "t2.small"
  }
```

```
fetch-values-variables.tf

resource "aws_instance" "myec2" {
   ami = "ami-082b5a644766e0e6f"
   instance_type = var.list[0]
}

variable "list" {
  type = list
  default = ["m5.large","m5.xlarge","t2.medium"]
}

variable "types" {
  type = map
  default = {
    us-east-1 = "t2.micro"
    us-west-2 = "t2.nano"
    ap-south-1 = "t2.small"
  }
```

Overview of Count Parameter

The count parameter on resources can simplify configurations and let you scale resources by simply incrementing a number.

Let's assume, you need to create two EC2 instances. One of the common approach is to define two separate resource blocks for aws_instance

```
resource "aws_instance" "instance-1" {
   ami = "ami-082b5a644766e0e6f"
   instance_type = "t2.micro"
}
```

+

```
resource "aws_instance" "instance-2" {
   ami = "ami-082b5a644766e0e6f"
   instance_type = "t2.micro"
}
```

```
count-parameter.tf

provider "aws" {
  region     = "us-west-2"
  access_key = "AKIAQIW66DN2W7WOYRGY"
  secret_key = "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"
}

  1
resource "aws_instance" "instance-1" {
   ami = "ami-082b5a644766e0e6f"
   instance_type = "t2.micro"
}
  2
resource "aws_instance" "instance-2" {
   ami = "ami-082b5a644766e0e6f"
   instance_type = "t2.micro"
}
```

Overview of Count Parameter

With count parameter, we can simply specify the count value and the resource can be scaled accordingly.

```
resource "aws_instance" "instance-1" {
   ami = "ami-082b5a644766e0e6f"
   instance_type = "t2.micro"
   count = 5
}
```

Understanding Challenge with Count

With the below code, terraform will create 5 IAM users. But the problem is that all will have the same name.

```
resource "aws_iam_user" "lb" {
  name = "loadbalancer"
  count = 5
  path = "/system/"
}
```

Adding $.{count.index} will create loadbalancers IAM users with different names starting with 01,02 etc

Understanding Challenge with Count

count.index allows us to fetch the index of each iteration in the loop.

```
resource "aws_iam_user" "lb" {
  name = "loadbalancer.${count.index}"
  count = 5
  path = "/system/"
}
```

```
Terraform will perform the following actions:

  # aws_iam_user.lb[0] will be created
  + resource "aws_iam_user" "lb" {
      + arn           = (known after apply)
      + force_destroy = false
      + id            = (known after apply)
      + name          = "loadbalancer.0"
      + path          = "/system/"
      + unique_id     = (known after apply)
    }

  # aws_iam_user.lb[1] will be created
  + resource "aws_iam_user" "lb" {
      + arn           = (known after apply)
      + force_destroy = false
      + id            = (known after apply)
      + name          = "loadbalancer.1"
      + path          = "/system/"
      + unique_id     = (known after apply)
    }
```

Understanding Challenge with Default Count Index

Having a username like loadbalancer0, loadbalancer1 might not always be suitable.

Better names like dev-loadbalancer, stage-loadbalancer, prod-loadbalancer is better.

count.index can help in such scenario as well.

```
variable "elb_names" {
  type        = list
  default     = ["dev-loadbalancer", "stage-loadbalancer", "prod-loadbalancer"]
}
```

```
provider "aws" {
  region     = "us-west-2"
  access_key = "AKIAQIW66DN2U5XYQS7L"
  secret_key = "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"
}

variable "elb_names" {
  type = list
  default = ["dev-loadbalancer", "stage-loadbalancer", "prod-loadbalancer"]
}

resource "aws_iam_user" "lb" {
  name = var.elb_names[count.index]
  count = 3
  path = "/system/"
```

Overview of Conditional Expression

A conditional expression uses the value of a bool expression to select one of two values.

Syntax of Conditional expression:

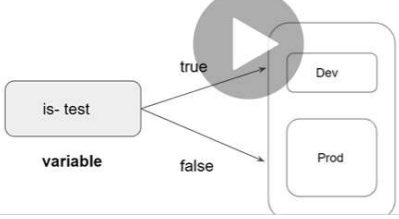
```
condition ? true_val : false_val
```

If condition is true then the result is true_val. If condition is false then the result is false_val.

Example of Conditional Expression

Let's assume that there are two resource blocks as part of terraform configuration

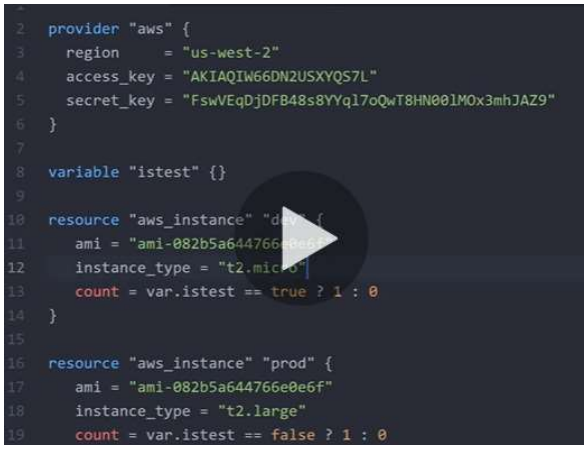
Depending on the variable value, one of the resource blocks will run.



In terraform.tfvars

Istest = true

In this condition when istest=true then 1 dev instance t2.micro is created
If you increase the count then that many instances are created



Overview of Conditional Expression

A conditional expression uses the value of a bool expression to select one of two values.

Syntax of Conditional expression:

```
condition ? true_val : false_val
```

If condition is true then the result is true_val. If condition is false then the result is false_val.

Overview of Local Values

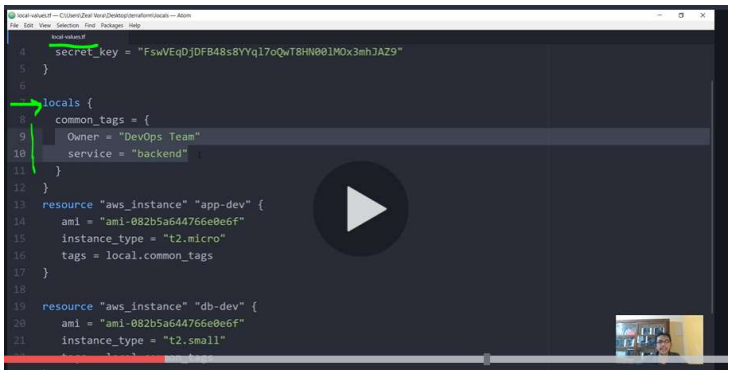
A local value assigns a name to an expression, allowing it to be used multiple times wi
module without repeating it.

```
locals {
  common_tags = {
    Owner = "DevOps Team"
    service = "backend"
  }
}
```

```
resource "aws_instance" "app-dev" {
  ami = "ami-082b5a644766e0e6f"
  instance_type = "t2.micro"
  tags = local.common_tags
}
```

```
resource "aws_ebs_volume" "db_ebs" {
  availability_zone = "us-west-2a"
  size              = 8
  tags = local.common_tags
```

First line of the code should be locals and then resource



Important Pointers for Local Values

Local values can be helpful to avoid repeating the same values or expressions multiple times in a configuration.

If overused they can also make a configuration hard to read by future maintainers by hiding the actual values used

Use local values only in moderation, in situations where a single value or result is used in many places and that value is likely to be changed in future.

Overview of Terraform Functions

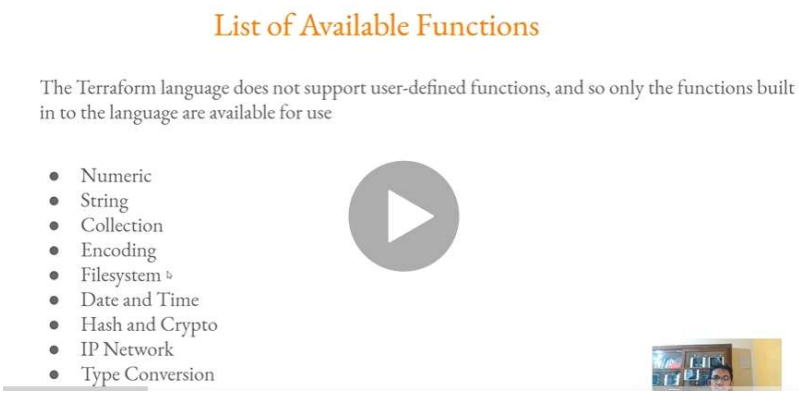
The Terraform language includes a number of built-in functions that you can use to transform and combine values.

The general syntax for function calls is a function name followed by comma-separated arguments in parentheses:

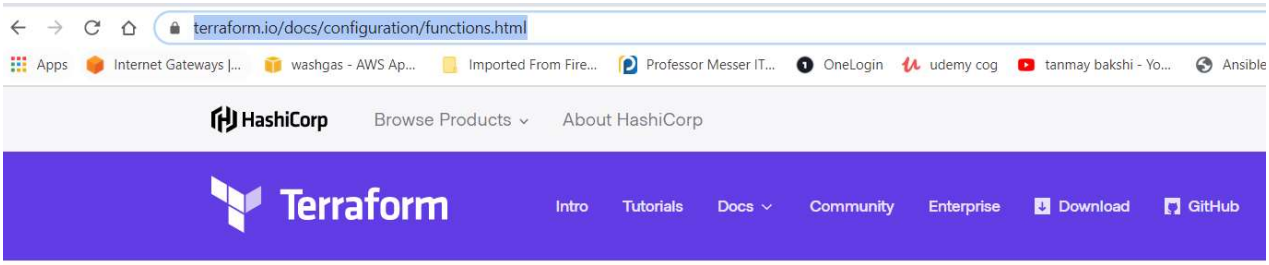
function (argument1, argument2)

Example:

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



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

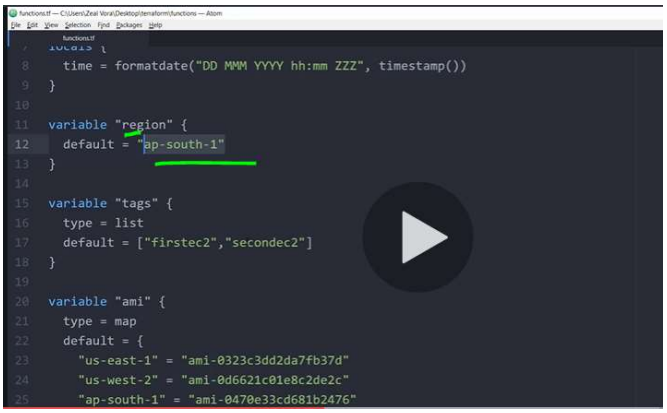


Here **a** value is called out and output is ay
There is no c value therefore output is what?

```
C:\Users\nkjaiswal\Desktop\Terraform>terraform console
>[J>

>[J> lookup({a="ay", b="bee"}, "a", "what?")
ay
>[J> lookup({a="ay", b="bee"}, "a", "what?")
ay
>[J> lookup({a="ay", b="bee"}, "b", "what?")
bee
>[J> lookup({a="ay", b="bee"}, "c", "what?")
what?
>[J>
```

variable region is **ap-south-1**
variable ami are listed below

Look up will check for region that is ap-south1 and select ami from ap-south1



```
  time = formatdate("DD MMM YYYY hh:mm ZZZ", timestamp())
}

variable "region" {
  default = "ap-south-1"
}

variable "tags" {
  type = list
  default = ["firstec2","secondec2"]
}

variable "ami" {
  type = map
  default = {
    "us-east-1" = "ami-0323c3dd2da7fb37d"
    "us-west-2" = "ami-0d6621c01e8c2de2c"
    "ap-south-1" = "ami-0470e33cd681b2476"
```

```
functions.tf
  default =  ap-south-1
}

variable "tags" {
  type = list
  default = ["firstec2","secondec2"]
}

variable "ami" {
  type = map
  default = {
    "us-east-1" = "ami-0323c3dd2da7fb37d"
    "us-west-2" = "ami-0d6621c01e8c2de2c"
    "ap-south-1" = "ami-0470e33cd681b2476"
  }
}

resource "aws_key_pair" "loginkey" {
  key_name   = "login-key"
  public_key = file("${path.module}/id_rsa.pub")
```

```
variable "ami" {
  type = map
  default = {
    "us-east-1" = "ami-0323c3dd2da7fb37d"
    "us-west-2" = "ami-0d6621c01e8c2de2c"
    "ap-south-1" = "ami-0470e33cd681b2476"
  }
}

resource "aws_key_pair" "loginkey" {
  key_name   = "login-key"
  public_key = file("${path.module}/id_rsa.pub")
}

resource "aws_instance" "app-dev" {
   ami = lookup(var.ami,var.region)
   instance_type = "t2.micro"
```

```
functions.tf
}

resource "aws_key_pair" "loginkey" {
  key_name   = "login-key"
  public_key = file("${path.module}/id_rsa.pub")
}

resource "aws_instance" "app-dev" {
   ami = lookup(var.ami,var.region)
   instance_type = "t2.micro"
   key_name = aws_key_pair.loginkey.key_name
   count = 2

   tags = {
     Name = element(var.tags,count.index)
   }
}
```

Here the instances created will have name -firstec2 and secondec2 beuz of element function

0 1 2

```
> element(["a", "b", "c"], 1)
b
> element(["a", "b", "c"], 0)
a
> element(["a", "b", "c"], 2)
c
>
```

```
   tags = {
     Name = element(var.tags,count.index)
   }
```

```
variable "tags" {
  type = list
  default = ["firstec2","secondec2"]
}
```

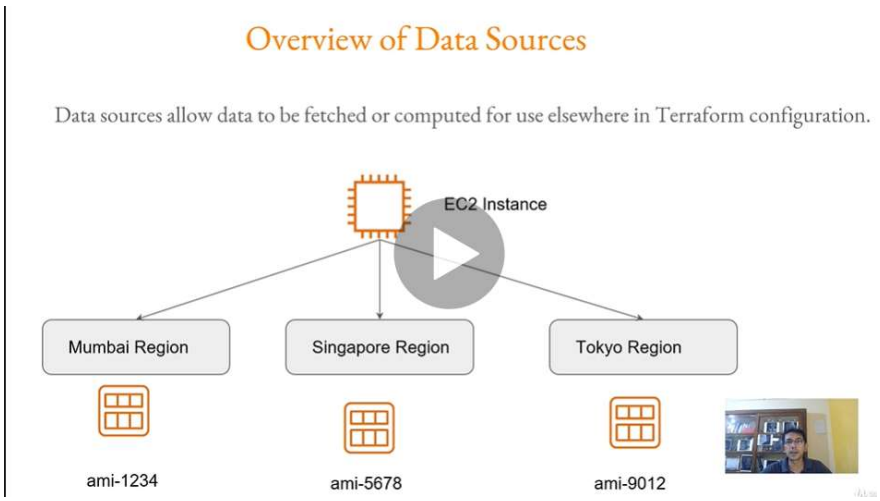
Here the instances are created with specific keypair

```
resource "aws_key_pair" "loginkey" {
  key_name   = "login-key"
  public_key = file("${path.module}/id_rsa.pub")
}
```

Public key

```
C:\Users\Zeal Vora\Desktop\terraform\functions>terraform console
[J> file("${path.module}/id_rsa.pub")
ssh-rsa AAAAB3NzaC1yc2EAAAADAQABAAABAQC8qO8KcNnKUm04ZC7H5s0WyJwpo/bxG/kJovGUqSz6ViEAhVxC9Tq/piJ9Kk9IUEOkfAjY8Yr5zn9ThR
bOVJ4AEHTjSwIie7YMMLjN+OdTn8+cqnfh9RNN3633ixGVP9CpbiDiB7gMsZ78Q2ps/gcxQuuW1X5t8Y0jcgHL0KJQsjU0eS7vhGCjRQ9snrgJxYg+UYM8
dOWINhbiVTQbydHGjcYUMZv6cWxZDQPyejObcFsmDY7UcD4ZnuzG/1VaSh+fXjNzqK6TjoY7ajH3F6WVW1Nbh6F/4hJipmT4Q5TxK51s28PCYveWZypc66
PTw2D1WHerCXQbuSnMlqpwip/f root@46400bafe371
```

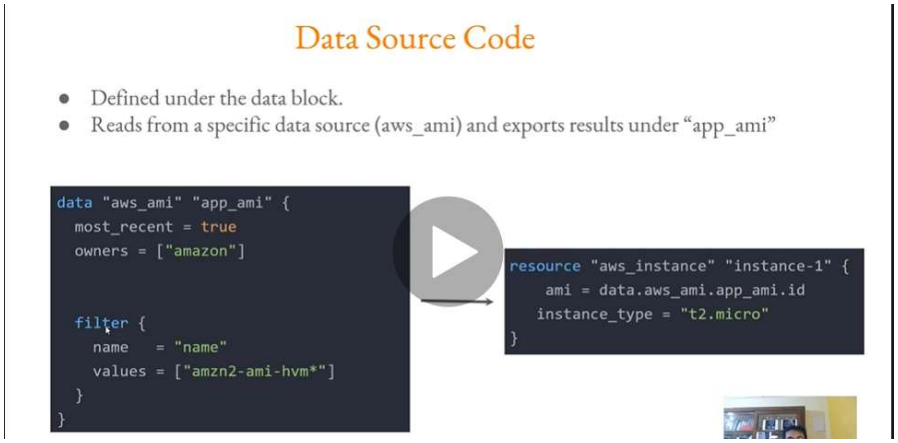
AMI ids are specific to region



Below error is shown in Terraform if wrong region is mentioned against specific AMI

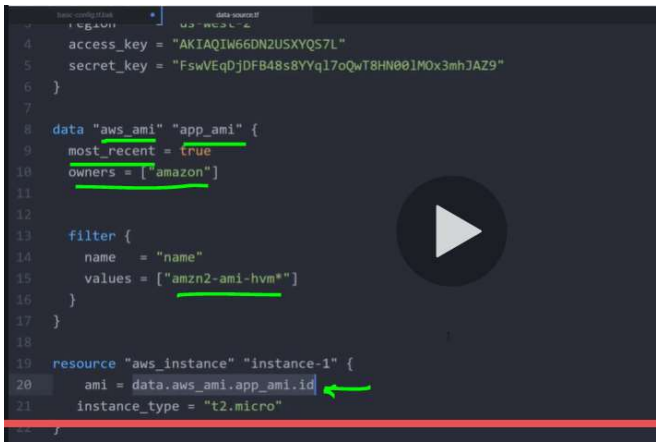
There will be no error seen when executing **terraform plan** but throws error in **terraform apply**

AMI id changes as per region in AWS

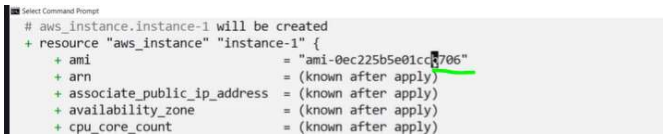


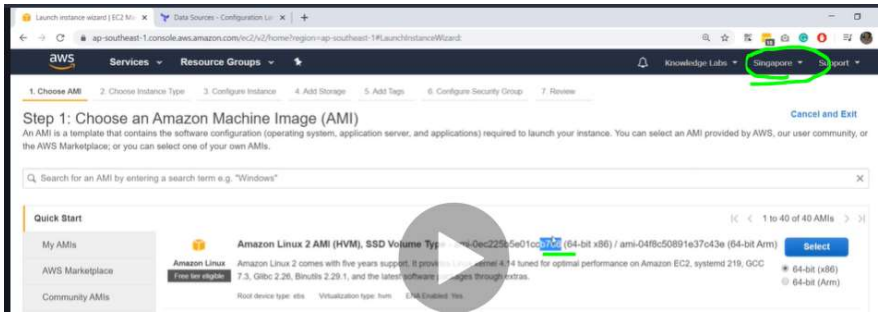
Here the data source is **"aws_ami"** which is passed to **"app_ami"**
It is most recent and owner is Amazon
Under Amazon name filter the AMI value

Ami = **data.aws_ami.app_ami.id**



Terraform picks the right ami id from Singapore region when executing terraform plan





Overview of Debugging Terraform

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

You can set TF_LOG to one of the log levels TRACE, DEBUG, INFO, WARN or ERROR to change the verbosity of the logs

```
bash-4.2# terraform plan
2020/04/22 13:45:31 [INFO] Terraform version: 0.12.24
2020/04/22 13:45:31 [INFO] Go runtime version: go1.12.13
2020/04/22 13:45:31 [INFO] CLI args: []string{"/usr/bin/terraform", "plan"}
2020/04/22 13:45:31 [DEBUG] Attempting to open CLI config file: /root/.terraformrc
2020/04/22 13:45:31 [DEBUG] File doesn't exist, but doesn't need to. Ignoring.
2020/04/22 13:45:31 [DEBUG] checking for credentials in "/root/.terraform.d/plugins"
2020/04/22 13:45:31 [INFO] CLI command args: []string{"plan"}
2020/04/22 13:45:31 [TRACE] Meta.Backend: built configuration for "s3" backend with hash value 789489680
2020/04/22 13:45:31 [TRACE] Meta.Backend: backend has not previously been initialized in this working directory
2020/04/22 13:45:31 [DEBUG] New state was assigned lineage "a10f92bf-686d-e6cf-3e9d-755be5c8a6a3"
2020/04/22 13:45:31 [TRACE] Meta.Backend: moving from default local state only to "s3" backend
```

In Linux server to enable Terraform log

```
Command Prompt - docker exec -it terraform bash
bash-4.2# export TF_LOG=TRACE
bash-4.2#
```

Detail logs are visible after enabling logging

```
2020/04/22 16:55:28 [TRACE] Executing graph transform *terraform.OrphanResourceCountTransformer
2020/04/22 16:55:28 [TRACE] vertex "aws_iam_user.lb (orphan)": evaluating
2020/04/22 16:55:28 [TRACE] Completed graph transform *terraform.OrphanResourceCountTransformer (no changes)
2020/04/22 16:55:28 [TRACE] Executing graph transform *terraform.AttachStateTransformer
2020/04/22 16:55:28 [TRACE] [walkPlan] Entering eval tree: aws_iam_user.lb (orphan)
2020/04/22 16:55:28 [TRACE] <root>: eval: *terraform.EvalSequence
2020/04/22 16:55:28 [DEBUG] Resource instance state not found for node "aws_instance.myec2", instance aws_instance.mye
c2
2020/04/22 16:55:28 [TRACE] <root>: eval: *terraform.EvalGetProvider
2020/04/22 16:55:28 [TRACE] Completed graph transform *terraform.AttachStateTransformer (no changes)
2020/04/22 16:55:28 [TRACE] <root>: eval: *terraform.EvalReadState
```

Specify location to store the logs

```
Command Prompt - docker exec -it terraform bash
bash-4.2# export TF_LOG_PATH=/tmp/terraform-crash.log
bash-4.2#
```

Important Pointers

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.

Importance of Readability

Anyone who is into programming knows the importance of formatting the code for readability.

The terraform fmt command is used to rewrite Terraform configuration files to take care of the overall formatting.

```
provider "aws" {
      region     = "us-west-2"
  access_key = "AKIAQIW66DN2W7WOYRGY"
  secret_key    = "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"
  version    = ">=2.10,<=2.30"
}
```

Before fmt

```
provider "aws" {
      region     = "us-west-2"
  access_key = "AKIAQIW66DN2W7WOYRGY"
  secret_key    = "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"
  version    = ">=2.10,<=2.30"
}
```

After fmt

```
provider "aws" {
  region     = "us-west-2"
  access_key = "AKIAQIW66DN2W7WOYRGY"
  secret_key = "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"
  version    = ">=2.10,<=2.30"
}
```

```
bash-4.2# cat test-fmt.tf
provider "aws" {
    region     = "us-west-2"
  access_key       = "AKIAQIW66DN2W7WOYRGY"
  secret_key = "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"
  version = ">=2.10,<=2.30"
}

resource "aws_instance" "myec2" {
  ami           = "ami-082b5a644766e0e6f"
  instance_type = "t2.micro"
}

bash-4.2# terraform fmt
test-fmt.tf
bash-4.2# cat test-fmt.tf
provider "aws" {
  region     = "us-west-2"
  access_key = "AKIAQIW66DN2W7WOYRGY"
  secret_key = "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"
  version    = ">=2.10,<=2.30"
}

resource "aws_instance" "myec2" {
  ami           = "ami-082b5a644766e0e6f"
```

Overview of Terraform Validate

Terraform Validate primarily checks whether a configuration is syntactically valid.

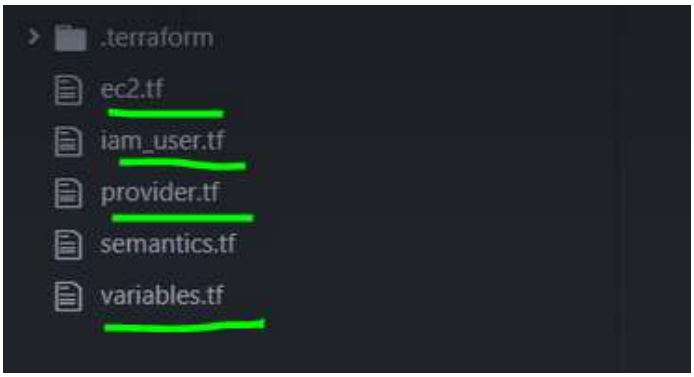
It can check various aspects including unsupported arguments, undeclared variables and others.

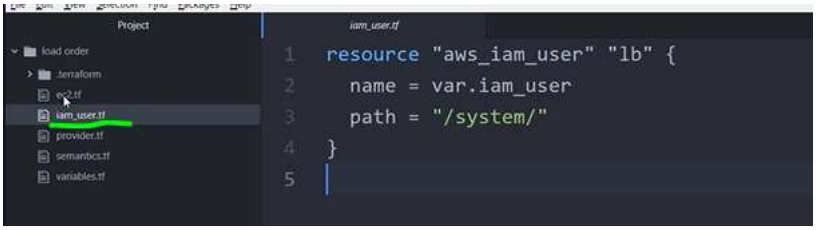
```
resource "aws_instance" "myec2" {
  ami           = "ami-082b5a644766e0e6f"
  instance_type = "t2.micro"
  sky = "blue"
}
```

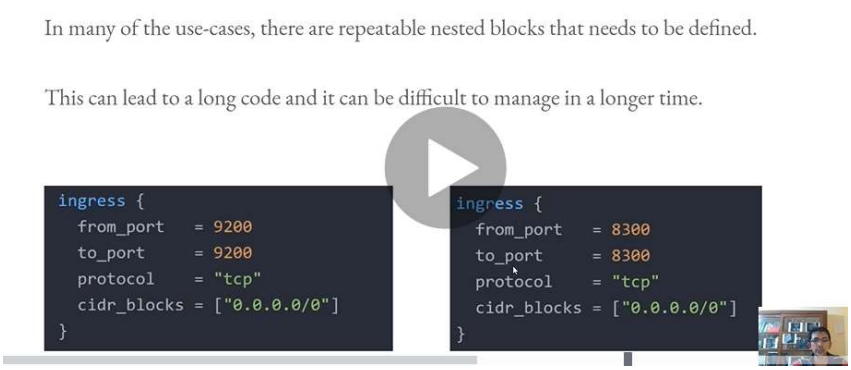
→

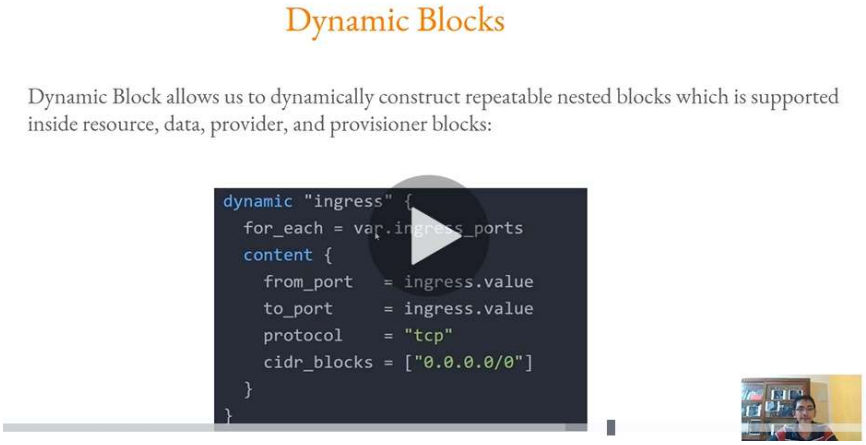
```
bash-4.2# terraform validate
Error: Unsupported argument
  on validate.tf line 10, in resource "aws_instance" "myec2":
  10:   sky = "blue"
An argument named "sky" is not expected here.
```

Create different terraform files separately for different categories so that it is easy to manage them separately for any addition or subraction









Before dynamic block

```
before.tf
resource "aws_security_group" "demo_sg" {
  name        = "sample-sg"

  ingress {
    from_port   = 8200
    to_port     = 8200
    protocol    = "tcp"
    cidr_blocks = ["0.0.0.0/0"]
  }

  ingress {
    from_port   = 8201
    to_port     = 8201
    protocol    = "tcp"
    cidr_blocks = ["0.0.0.0/0"]
  }

  ingress {
    from_port   = 8300
    to_port     = 8300
    protocol    = "tcp"
```

After Dynamic block

```
variable "sg_ports" {
  type        = list(number)
  description = "list of ingress ports"
  default     = [8200, 8201,8300, 9200, 9500]
}

resource "aws_security_group" "dynamicsg" {
  name        = "dynamic-sg"
  description = "Ingress for Vault"

  dynamic "ingress" {
    for_each = var.sg_ports
    content {
      from_port   = ingress.value
      to_port     = ingress.value
      protocol    = "tcp"
      cidr_blocks = ["0.0.0.0/0"]
    }
```

iterator

```
  description = "list of ingress ports"
  default     = [8200, 8201,8300, 9200, 9500]
}

resource "aws_security_group" "dynamicsg" {
  name        = "dynamic-sg"
  description = "Ingress for Vault"

  dynamic "ingress" {
    for_each = var.sg_ports
    iterator = port
    content {
      from_port   = port.value
      to_port     = port.value
      protocol    = "tcp"
      cidr_blocks = ["0.0.0.0/0"]
    }
  }
```

Iterators

The iterator argument (optional) sets the name of a temporary variable that represents the current element of the complex value

If omitted, the name of the variable defaults to the label of the dynamic block ("ingress" in the example above).

```
dynamic "ingress" {
  for_each = var.ingress_ports
  content {
    from_port   = ingress.value
    to_port     = ingress.value
    protocol    = "tcp"
    cidr_blocks = ["0.0.0.0/0"]
  }
}
```

→

```
dynamic "ingress" {
  for_each = var.ingress_ports
  iterator = port
  content {
    from_port   = port.value
    to_port     = port.value
    protocol    = "tcp"
    cidr_blocks = ["0.0.0.0/0
  }
```

Provisioners are interesting

Till now we have been working only on creation and destruction of infrastructure scenarios.

Let's take an example:

We created a web-server EC2 instance with Terraform.

Problem: It is only an EC2 instance, it does not have any software installed.

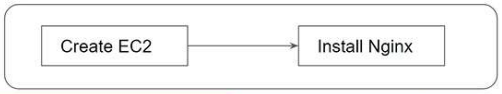
What if we want a complete end to end solution ?

Welcome to Terraform Provisioners

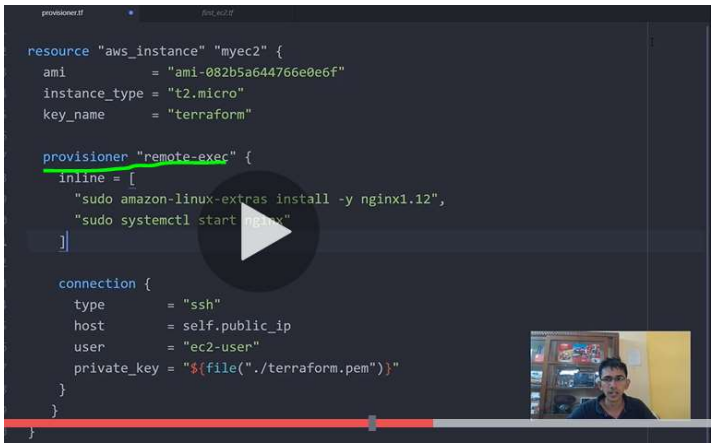
Provisioners are used to execute scripts on a local or remote machine as part of resource creation or destruction.

Let's take an example:

On creation of Web-Server, execute a script which installs Nginx web-server.



Create EC2 instance and install NGNIX software | Start the services

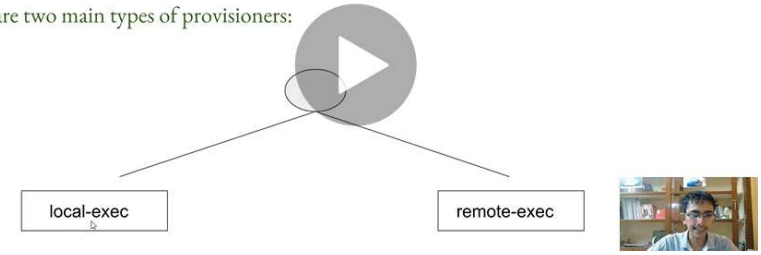


```
provisioner.tf        first_ec2.tf

resource "aws_instance" "myec2" {
  ami           = "ami-082b5a644766e0e6f"
  instance_type = "t2.micro"
  key_name      = "terraform"

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

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

Provisioners are interesting

Terraform has capability to turn provisioners both at the time of resource creation as well as destruction.

There are two main types of provisioners:

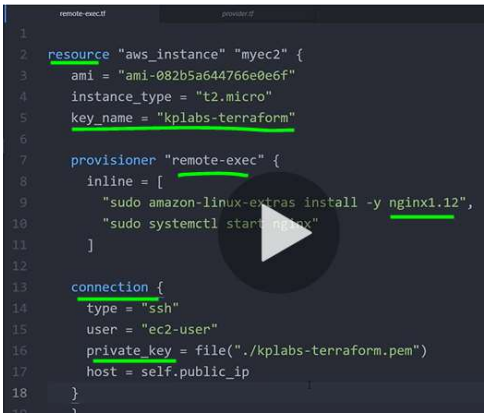


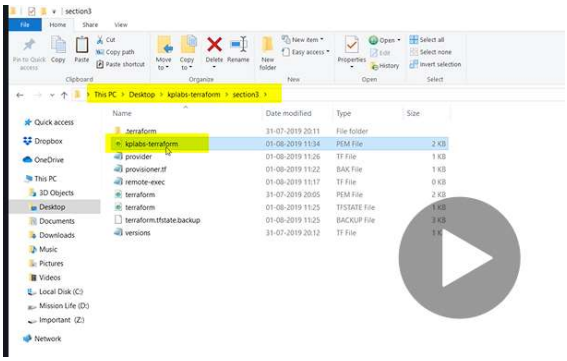
https://www.terraform.io/docs/provisioners/index.html

https://www.terraform.io/docs/provisioners/remote-exec.html

Resource section Key_name is the name of the pem file
Provisioner install ngnix in linux sever
Connection private key file is the location of key file stored in Terraform file location
Self-public ip is attribute of EC2 instance to get the ip of the instance

Make changes in security group for Terraform to login to EC2 instance





Terraform Plan
Terraform apply

```
aws_instance.myec2: Creating...
aws_instance.myec2: Still creating... [10s elapsed]
aws_instance.myec2: Still creating... [20s elapsed]
aws_instance.myec2: Still creating... [30s elapsed]
aws_instance.myec2: Still creating... [40s elapsed]
aws_instance.myec2: Provisioning with 'remote-exec'...
aws_instance.myec2 (remote-exec): Connecting to remote host via SSH...
aws_instance.myec2 (remote-exec):   Host: 34.208.101.248
aws_instance.myec2 (remote-exec):   User: ec2-user
aws_instance.myec2 (remote-exec):   Password: false
aws_instance.myec2 (remote-exec):   Private key: true
aws_instance.myec2 (remote-exec):   Certificate: false
aws_instance.myec2 (remote-exec):   SSH Agent: false
aws_instance.myec2 (remote-exec):   Checking Host Key: false
aws_instance.myec2 (remote-exec): Connected!
aws_instance.myec2: Still creating... [50s elapsed]
aws_instance.myec2 (remote-exec): Installing nginx
aws_instance.myec2 (remote-exec): Loaded plugins: extras_suggestions,
aws_instance.myec2 (remote-exec):               : langpacks, priorities,
aws_instance.myec2 (remote-exec):               : update-motd
```

Local exe is executing the terraform code is the same server after it is created.

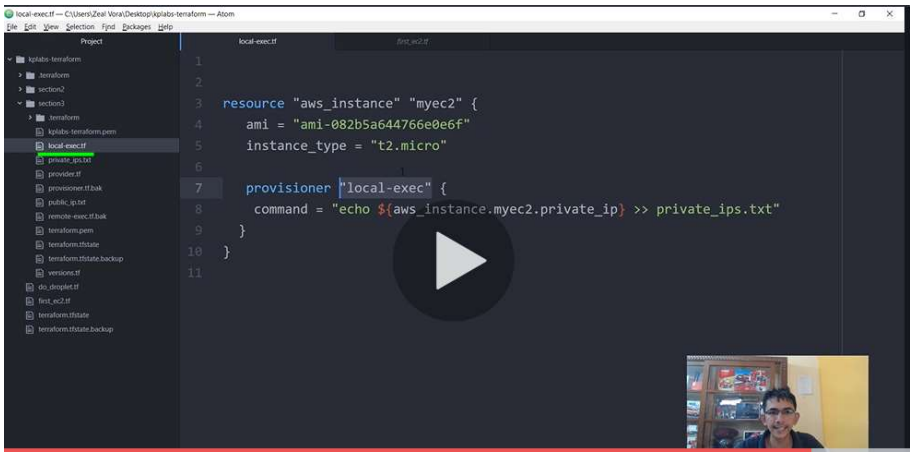
Provisioners are interesting

local-exec provisioners allows us to invoke a local executable after the resource is created.

One of the most used approach of local-exec is to run ansible-playbooks on the created server after the resource is created.

```
provisioner "local-exec" {
  command = "echo ${aws_instance.web.private_ip} >> private_ips.txt"
}
```

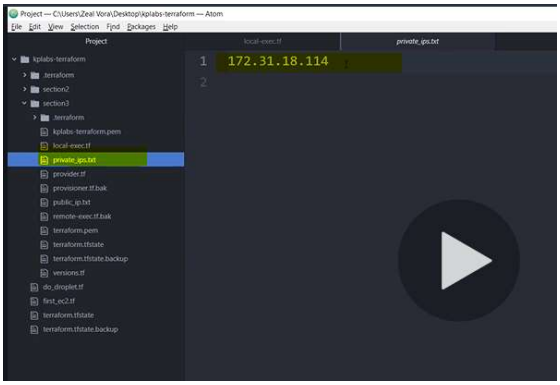
This is sample code which creates EC2 instance and stores the private ips in local machine where code is executed.



After EC2 instance is created …local-exec is executed and private ip is displayed

```
Do you want to perform these actions?
  Terraform will perform the actions described above.
  Only 'yes' will be accepted to approve.

  Enter a value: yes

aws_instance.myec2: Creating...
aws_instance.myec2: Still creating... [10s elapsed]
aws_instance.myec2: Still creating... [20s elapsed]
aws_instance.myec2: Still creating... [30s elapsed]
aws_instance.myec2: Still creating... [40s elapsed]
aws_instance.myec2: Provisioning with 'local-exec'...
aws_instance.myec2 (local-exec): Executing: ["cmd" "/C" "echo 172.31.18.114 >> private_ips.
aws_instance.myec2: Creation complete after 45s [id=i-068a72b5738fdaf72]
```

Private ip is stored in local computer



Overview of Provisioner Types

There are two primary types of provisioners:

Types of Provisioners	Description
Creation-Time Provisioner	Creation-time provisioners are only run during creation, not during updating or any other lifecycle If a creation-time provisioner fails, the resource is marked as tainted.
Destroy-Time Provisioner	Destroy provisioners are run before the resource is destroyed.

Destroy Time Provisioner

If when = destroy is specified, the provisioner will run when the resource it is defined within is destroyed.

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

  provisioner "local-exec" {
    when    = destroy
    command = "echo 'Destroy-time provisioner'"
  }
}
```

When EC2 instance is created it install the nano package

When EC2 instance is deleted it first uninstalls the nano package and then instance is deleted.

```
provisioner "remote-exec" {
  inline = [
    "sudo yum -y install nano"
  ]
}
provisioner "remote-exec" {
    when    = destroy
    inline = [
      "sudo yum -y remove nano"
    ]
  }
connection {
  type = "ssh"
  user = "ec2-user"
  private_key = file("./ec2-key.pem")
  host = self.public_ip
}
```

Security group and EC2 instance is created and then nano package is installed in same EC2 instance.

```
Select Command Prompt

C:\Users\Zeal Vora\Desktop\terraform\provisioner types>terraform apply -auto-approve
aws_security_group.allow_ssh: Creating...
aws_security_group.allow_ssh: Creation complete after 3s [id=sg-0bc1e50a2682d3d34]
aws_instance.myec2: Creating...
aws_instance.myec2: Still creating... [10s elapsed]
aws_instance.myec2: Still creating... [20s elapsed]
aws_instance.myec2: Provisioning with 'remote-exec'...
aws_instance.myec2 (remote-exec): Connecting to remote host via SSH...
aws_instance.myec2 (remote-exec):   Host: 54.251.157.13
```

```
aws_instance.myec2: Provisioning with 'remote-exec'...
aws_instance.myec2 (remote-exec): Connecting to remote
```

```
aws_instance.myec2 (remote-exec): Package nano-2.9.8-2.amzn2.0.1.x86_64 already installed and latest version
```

To destroy

First removes the nano package and then EC2 instance is destroyed with **terraform destroy** command

```
C:\Users\Zeal Vora\Desktop\terraform\provisioner types>terraform destroy -auto-approve
```

```
Select Command Prompt

C:\Users\Zeal Vora\Desktop\terraform\provisioner types>terraform destroy -auto-approve
aws_security_group.allow_ssh: Refreshing state... [id=sg-0bc1e50a2682d3d34]
aws_instance.myec2: Refreshing state... [id=i-0492d9ce5e5d966e5]
aws_instance.myec2: Destroying... [id=i-0492d9ce5e5d966e5]
aws_instance.myec2: Provisioning with 'remote-exec'...
aws_instance.myec2 (remote-exec): Connecting to remote host via SSH...
aws_instance.myec2 (remote-exec):   Host: 54.251.157.13
```

```
Select Command Prompt
aws_instance.myec2 (remote-exec): Loaded plugins: extras_suggestions,
aws_instance.myec2 (remote-exec):                : langpacks, priorities,
aws_instance.myec2 (remote-exec):                : update-motd
aws_instance.myec2 (remote-exec): Resolving Dependencies
aws_instance.myec2 (remote-exec): --> Running transaction check
aws_instance.myec2 (remote-exec): ---> Package nano.x86_64 0:2.9.8-2.amzn2.0.1 will be erased
aws_instance.myec2 (remote-exec): --> Finished Dependency Resolution

aws_instance.myec2 (remote-exec): Dependencies Resolved

aws_instance.myec2 (remote-exec): ==========================================
aws_instance.myec2 (remote-exec):  Package
aws_instance.myec2 (remote-exec):       Arch   Version    Repository
aws_instance.myec2 (remote-exec):                                   Size
aws_instance.myec2 (remote-exec): ==========================================
aws_instance.myec2 (remote-exec): Removing:
aws_instance.myec2 (remote-exec):  nano x86_64 2.9.8-2.amzn2.0.1
aws_instance.myec2 (remote-exec):                                  installed 2.2 M

aws_instance.myec2 (remote-exec): Transaction Summary
aws_instance.myec2 (remote-exec): ==========================================
aws_instance.myec2 (remote-exec): Remove  1 Package
```

```
Select Command Prompt
aws_instance.myec2 (remote-exec): Removing:
aws_instance.myec2 (remote-exec):  nano x86_64 2.9.8-2.amzn2.0.1
aws_instance.myec2 (remote-exec):                        installed 2.2 M

aws_instance.myec2 (remote-exec): Transaction Summary
aws_instance.myec2 (remote-exec): ==========================================
aws_instance.myec2 (remote-exec): Remove  1 Package

aws_instance.myec2 (remote-exec): Installed size: 2.2 M
aws_instance.myec2 (remote-exec): Downloading packages:
aws_instance.myec2 (remote-exec): Running transaction check
aws_instance.myec2 (remote-exec): Running transaction test
aws_instance.myec2 (remote-exec): Transaction test succeeded
aws_instance.myec2 (remote-exec): Running transaction
aws_instance.myec2 (remote-exec):   Erasing    : nano-2.9.8-2.amzn2   1/1
aws_instance.myec2 (remote-exec):   Verifying  : nano-2.9.8-2.amzn2   1/1

aws_instance.myec2 (remote-exec): Removed:
aws_instance.myec2 (remote-exec):   nano.x86_64 0:2.9.8-2.amzn2.0.1

aws_instance.myec2 (remote-exec): Complete!
aws_instance.myec2: Still destroying... [id=i-0492d9ce5e5d966e5, 10s elapsed]
aws_instance.myec2: Still destroying... [id=i-0492d9ce5e5d966e5, 20s elapsed]
aws_instance.myec2: Still destroying... [id=i-0492d9ce5e5d966e5, 30s elapsed]
aws_instance.myec2: Destruction complete after 34s
```

Provisioner - Failure Behaviour

By default, provisioners that fail will also cause the Terraform apply itself to fail.

The on_failure setting can be used to change this. The allowed values are:

Allowed Values	Description
continue	Ignore the error and continue with creation or destruction.
fail	Raise an error and stop applying (the default behavior). If this is a creation provisioner, taint the resource.

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

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

In this security group is created with inbound rule and also creates EC2 instance and installs nano

```
failure-behavior.tf

resource "aws_security_group" "allow_ssh" {
  name        = "allow_ssh"
  description = "Allow SSH inbound traffic"

  ingress {
    description = "SSH into VPC"
    from_port   = 22
    to_port     = 22
    protocol    = "tcp"
    cidr_blocks = ["0.0.0.0/0"]
  }
}

resource "aws_instance" "myec2" {
    ami = "ami-0b1e534a4ff9019e0"
    instance_type = "t2.micro"
    key_name = "ec2-key"
    vpc_security_group_ids  = [aws_security_group.allow_ssh.id]
```

```
failure-behavior.tf — C:\Users\Zeal Vora\Desktop\terraform — Atom
File  Edit  View  Selection  Find  Packages  Help
failure-behavior.tf

}

resource "aws_instance" "myec2" {
    ami = "ami-0b1e534a4ff9019e0"
    instance_type = "t2.micro"
    key_name = "ec2-key"
    vpc_security_group_ids  = [aws_security_group.allow_ssh.id]

    provisioner "remote-exec" {
      inline = [
        "sudo yum -y install nano"
      ]
    }
    connection {
      type = "ssh"
      user = "ec2-user"
      private_key = file("./ec2-key.pem")
      host = self.public_ip
```

Ingress rule is removed and also it is known that nano package will fail to install.

Here it is seen that EC2 instance is created but nano package is not installed but terraform in next apply will be either destroyed or re-created

```
Command Prompt - terraform  apply -auto-approve
C:\Users\Zeal Vora\Desktop\terraform\provisioner-failure-behaviour>terraform apply -auto-approve
aws_security_group.allow_ssh: Creating...
aws_security_group.allow_ssh: Creation complete after 3s [id=sg-01e05b8626d4ee796]
aws_instance.myec2: Creating...
aws_instance.myec2: Still creating... [10s elapsed]
aws_instance.myec2: Still creating... [20s elapsed]
aws_instance.myec2: Provisioning with 'remote-exec'...
aws_instance.myec2 (remote-exec): Connecting to remote host via SSH...
aws_instance.myec2 (remote-exec):   Host: 13.228.27.63
aws_instance.myec2 (remote-exec):   User: ec2-user
aws_instance.myec2 (remote-exec):   Password: false
aws_instance.myec2 (remote-exec):   Private key: true
aws_instance.myec2 (remote-exec):   Certificate: false
aws_instance.myec2 (remote-exec):   SSH Agent: false
aws_instance.myec2 (remote-exec):   Checking Host Key: false
```

```
aws_instance.myec2 (remote-exec):            yum-config-manager --save --setopt=<repoi
aws_instance.myec2 (remote-exec): Cannot find a valid baseurl for repo: amzn2-core/2/x8

Error: error executing "/tmp/terraform_1691779483.sh": Process exited with status 1
```

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

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

Allowed Values	Description
continue	Ignore the error and continue with creation or destruction.

If **on_failure = continue** is added then even if the nano package is not installed in EC2 instance but terraform will not recreate or destroy the EC2 instance in next apply

```
    to_port     = 22
    protocol    = "tcp"
    cidr_blocks = ["0.0.0.0/0"]
  }
}

resource "aws_instance" "myec2" {
   ami = "ami-0b1e534a4ff9019e0"
   instance_type = "t2.micro"
   key_name = "ec2-key"
   vpc_security_group_ids  = [aws_security_group.allow_ssh.id]

   provisioner "remote-exec" {
     on_failure = continue
     inline = [
       "sudo yum -y install nano"
     ]
   }
   connection {
     type = "ssh"
```

```
Command Prompt
C:\Users\Zeal Vora\Desktop\terraform\provisioner-failure-behaviour>terraform apply -auto-approve
```

```
aws_instance.myec2 (remote-exec):         so will have to try and fail each time (and thus. yum will be be much
aws_instance.myec2 (remote-exec):         slower). If it is a very temporary problem though, this is often a nice
aws_instance.myec2 (remote-exec):         compromise:

aws_instance.myec2 (remote-exec):             yum-config-manager --save --setopt=<repoid>.skip_if_unavailable=true

aws_instance.myec2 (remote-exec): Cannot find a valid baseurl for repo: amzn2-core/2/x86_64

aws_instance.myec2: Creation complete after 1m59s [id=i-0735017c17b95a145]

Apply complete! Resources: 2 added, 0 changed, 0 destroyed.
```

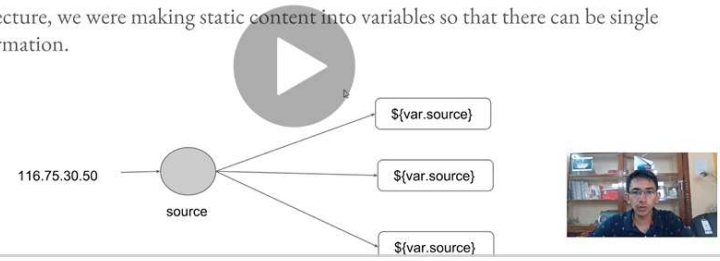
So it is seen that next terraform apply there are no changes

```
C:\Users\Zeal Vora\Desktop\terraform\provisioner-failure-behaviour>terraform plan
Refreshing Terraform state in-memory prior to plan...
The refreshed state will be used to calculate this plan, but will not be
persisted to local or remote state storage.

aws_security_group.allow_ssh: Refreshing state... [id=sg-07cfcf1fb87e0b55a]
aws_instance.myec2: Refreshing state... [id=i-0735017c17b95a145]

------------------------------------------------------------------------

No changes. Infrastructure is up-to-date.

This means that Terraform did not detect any differences between your
configuration and real physical resources that exist. As a result, no
actions need to be performed.

C:\Users\Zeal Vora\Desktop\terraform\provisioner-failure-behaviour>
```

Understanding DRY Approach

In software engineering, don't repeat yourself (DRY) is a principle of software development aimed at reducing repetition of software patterns.

In the earlier lecture, we were making static content into variables so that there can be single source of information.



We are repeating resource code

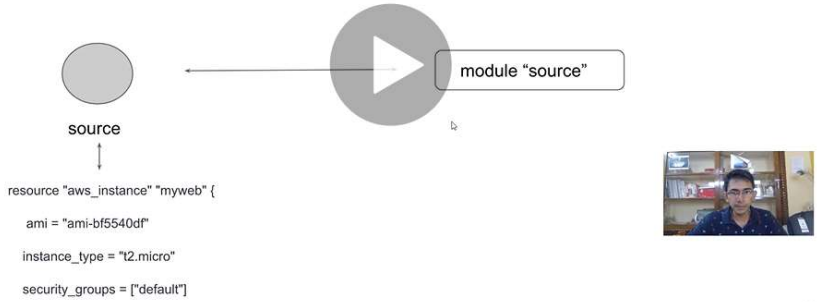
We do repeat multiple times various terraform resources for multiple projects.

Sample EC2 Resource

```
resource "aws_instance" "myweb" {
  ami = "ami-bf5540df"
  instance_type = "t2.micro"
  security_groups = ["default"]
}
```

Centralized Structure

We can centralize the terraform resources and can call out from TF files whenever required.



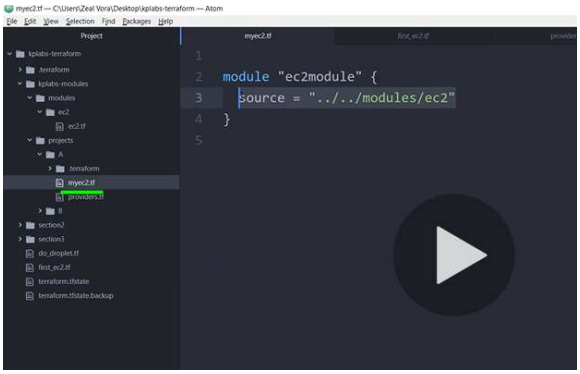
Source

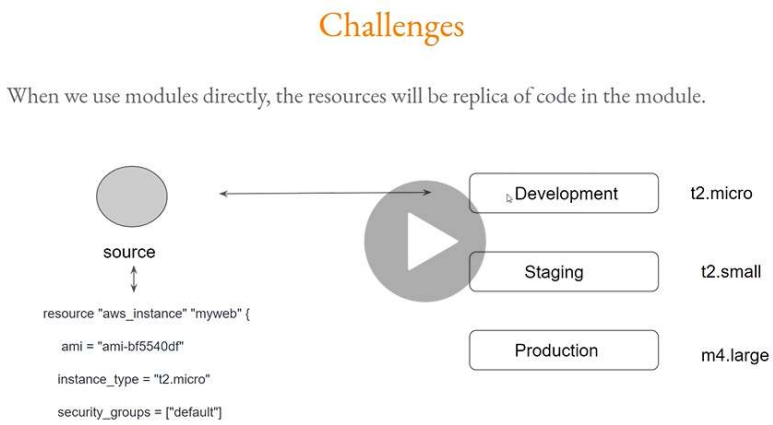
Actual code in file first_ec2.tf

Module source

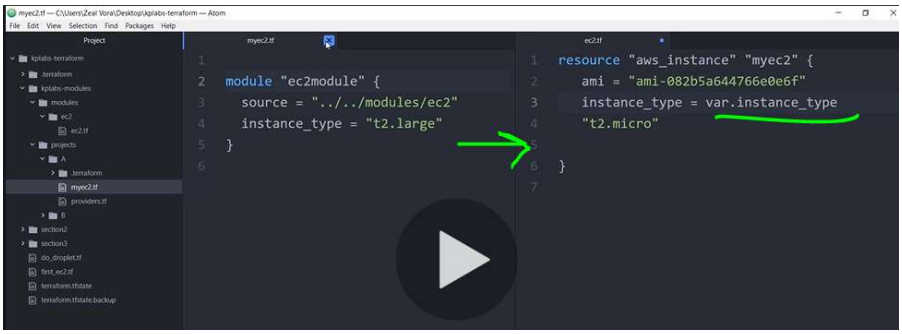
Develops can use this to create EC2 instances without knowing details about it.

Same source code is not repeated every time when EC2 instances are created





Create a variable for EC2 type



Overview of Terraform Registry

The Terraform Registry is a repository of modules written by the Terraform community.

The registry can help you get started with Terraform more quickly

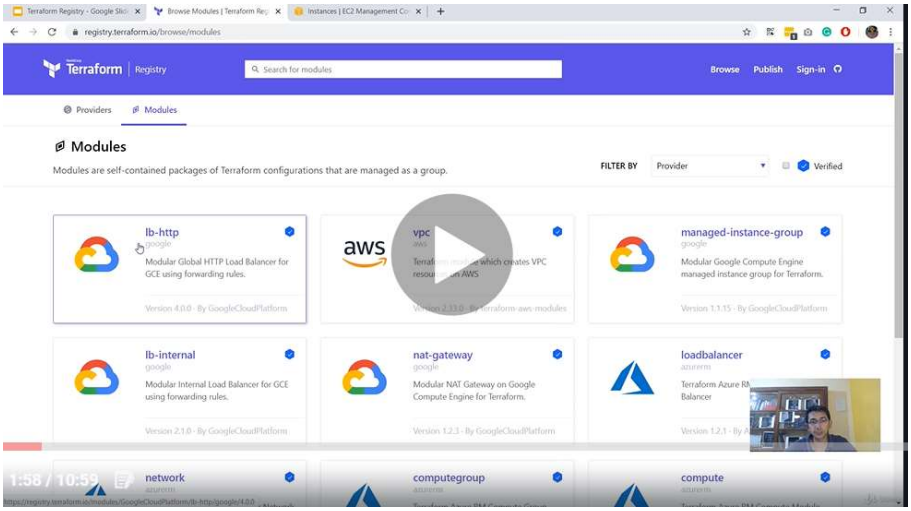
Verified Modules in Terraform Registry

Verified modules are reviewed by HashiCorp and actively maintained by contributors to stay up-to-date and compatible with both Terraform and their respective providers.

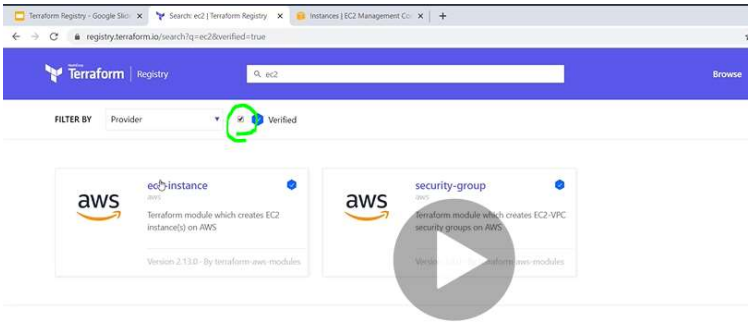
The blue verification badge appears next to modules that are verified.

Module verification is currently a manual process restricted to a small group of trusted HashiCorp partners.

Modules for different providers



Verified modules which are certified by Terraform

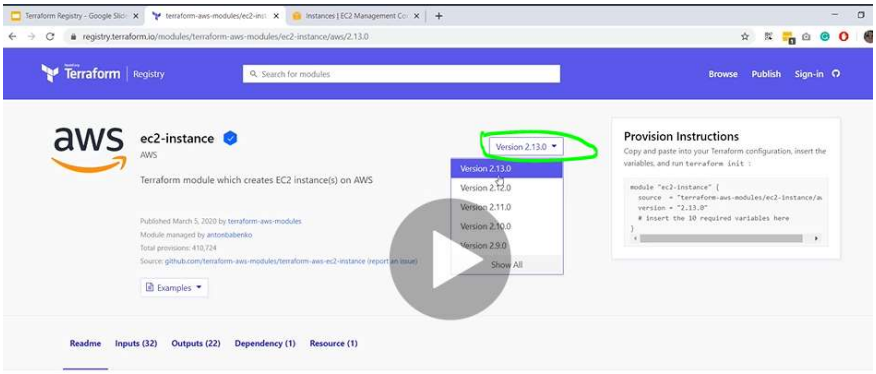


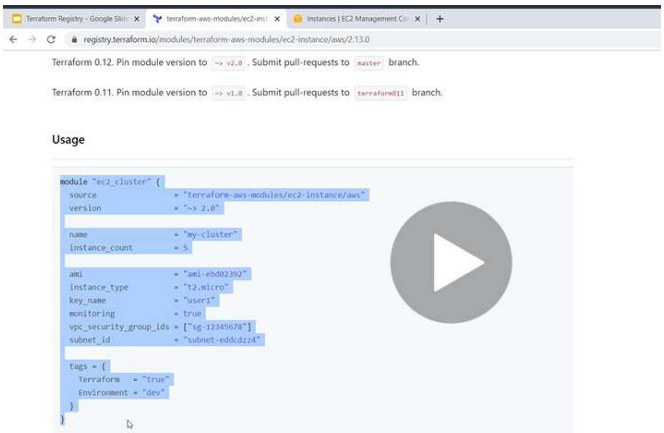
Using Registry Module in Terraform

To use Terraform Registry module within the code, we can make use of the source argument that contains the module path.

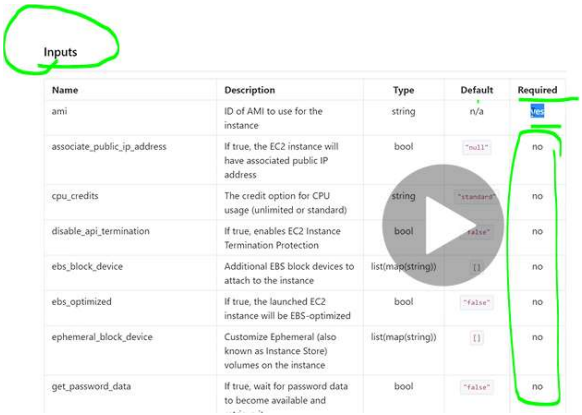
Below code references to the EC2 Instance module within terraform registry.

```
module "ec2-instance" {
  source  = "terraform-aws-modules/ec2-instance/aws"
  version = "2.13.0"
  # insert the 10 required variables here
}
```





Check the inputs which are required that is put in terraform script



Inputs

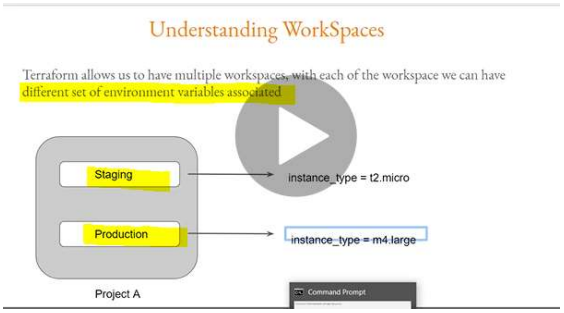
Name	Description	Type	Default	Required
ami	ID of AMI to use for the instance	string	n/a	yes
associate_public_ip_address	If true, the EC2 instance will have associated public IP address	bool	"null"	no
cpu_credits	The credit option for CPU usage (unlimited or standard)	string	"standard"	no
disable_api_termination	If true, enables EC2 Instance Termination Protection	bool	"false"	no
ebs_block_device	Additional EBS block devices to attach to the instance	list(map(string))	[]	no
ebs_optimized	If true, the launched EC2 instance will be EBS-optimized	bool	"false"	no
ephemeral_block_device	Customize Ephemeral (also known as Instance Store) volumes on the instance	list(map(string))	[]	no
get_password_data	If true, wait for password data to become available and	bool	"false"	no

```
Select Command Prompt - docker exec -it terraform bash
bash-4.2# ls -la
total 16
drwxr-xr-x 3 root root 4096 Apr 23 10:30 .
dr-xr-x--- 1 root root 4096 Apr 23 08:22 ..
drwxr-xr-x 4 root root 4096 Apr 23 09:51 .terraform
-rw-r--r-- 1 root root  539 Apr 23 10:30 registry-terraform.tf
bash-4.2# cd .terraform/
bash-4.2# ls
modules  plugins  terraform.tfstate
bash-4.2# cd modules/
ec2_cluster/  modules.json
bash-4.2# cd modules/ec2_cluster/terraform-aws-ec2-instance-2.13.0/
bash-4.2# ls
CHANGELOG.md  LICENSE  Makefile  README.md  examples  main.tf  outputs.tf  variables.tf
bash-4.2#
```

```
CHANGELOG.md  LICENSE  Makefile  README.md  examples  main.tf  outputs.tf  variables.tf
bash-4.2# nano main.tf
```

Main.tf

```
Command Prompt - docker exec -it terraform bash
  GNU nano 2.9.8                                   main.tf

locals {
  is_t_instance_type = replace(var.instance_type, "/^t(2|3|3a){1}\\..*$/", "1") == "1" ? true : false
}

resource "aws_instance" "this" {
  count = var.instance_count

  ami              = var.ami
  instance_type    = var.instance_type
  user_data        = var.user_data
  user_data_base64 = var.user_data_base64
  subnet_id = length(var.network_interface) > 0 ? null : element(
    distinct(compact(concat([var.subnet_id], var.subnet_ids)))
    count.index,
  )
  key_name               = var.key_name
  monitoring             = var.monitoring
  get_password_data      = var.get_password_data
  vpc_security_group_ids = var.vpc_security_group_ids
  iam_instance_profile   = var.iam_instance_profile

  associate_public_ip_address = var.associate_public_ip_address
  private_ip                  = length(var.private_ips) > 0 ? element(var.private_ips, count.index)
```



Commands

Terraform workspace
Terraform workspace show
Terraform workspace list
Terraform workspace select dev

```
C:\Users\Zeal Vora\Desktop\kplabs-workspaces\projects\A>terraform workspace list
  default
  dev
* prd
```

```
Command Prompt - terraform plan

C:\Users\Zeal Vora\Desktop\kplabs-workspaces\projects\A>terraform workspace
Usage: terraform workspace

  Create, change and delete Terraform workspaces.

C:\Users\Zeal Vora\Desktop\kplabs-workspaces\projects\A>terraform workspace show
prd

C:\Users\Zeal Vora\Desktop\kplabs-workspaces\projects\A>terraform plan
Refreshing Terraform state in-memory prior to plan...
The refreshed state will be used to calculate this plan, but will not be
persisted to local or remote state storage.

------------------------------------------------------------------------
```

In production workspace instance type created is m4.large

```
+ module.myec2.aws_instance.myweb
    id:                          <computed>
    ami:                         "ami-bf5540df"
    associate_public_ip_address: <computed>
    availability_zone:           <computed>
    ebs_block_device.#:          <computed>
    ephemeral_block_device.#:    <computed>
    get_password_data:           "false"
    instance_state:              <computed>
    instance_type:               "m4.large"
    ipv6_address_count:          <computed>
```

In dev workspace instance type created is t2.micro

```
C:\Users\Zeal Vora\Desktop\kplabs-workspaces\projects\A>terraform workspace select dev
```

```
Select Command Prompt
+ module.myec2.aws_instance.myweb
    id:                          <computed>
    ami:                         "ami-bf5540df"
    associate_public_ip_address: <computed>
    availability_zone:           <computed>
    ebs_block_device.#:          <computed>
    ephemeral_block_device.#:    <computed>
    get_password_data:           "false"
    instance_state:              <computed>
    instance_type:               "t2.micro"
    ipv6_address_count:          <computed>
```

How to create workspace and define code in terraform

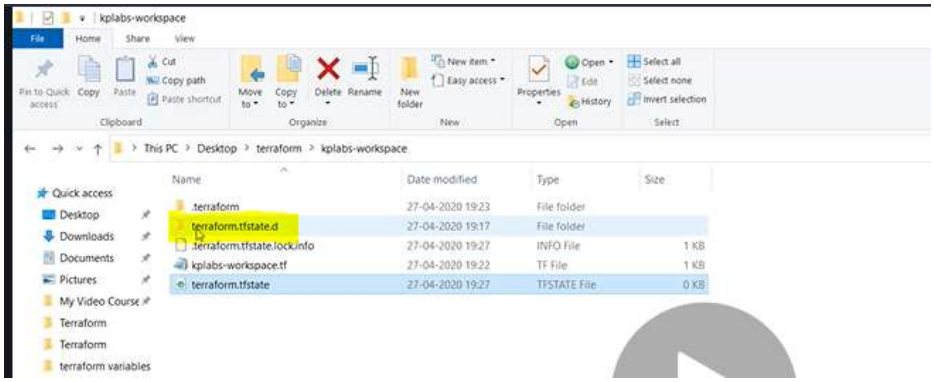
```
C:\Users\Zeal Vora\Desktop\terraform\kplabs-workspace>terraform workspace -h
Usage: terraform workspace

  New, list, select and delete Terraform workspaces.

Subcommands:
    delete    Delete a workspace
    list      List Workspaces
    new       Create a new workspace
    select    Select a workspace
    show      Show the name of the current workspace

C:\Users\Zeal Vora\Desktop\terraform\kplabs-workspace>terraform workspace show
default

C:\Users\Zeal Vora\Desktop\terraform\kplabs-workspace>terraform workspace new dev
Created and switched to workspace "dev"!

You're now on a new, empty workspace. Workspaces isolate their state,
so if you run "terraform plan" Terraform will not see any existing state
for this configuration.

C:\Users\Zeal Vora\Desktop\terraform\kplabs-workspace>terraform workspace show
dev
```

```
C:\Users\Zeal Vora\Desktop\terraform\kplabs-workspace>terraform workspace new prd
Created and switched to workspace "prd"!
```

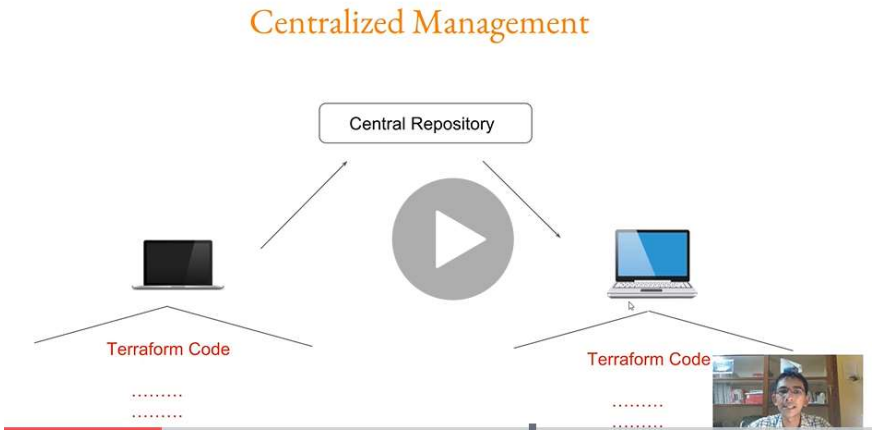
```
kplabs-workspace.tf
provider "aws" {
  region     = "us-west-2"
  access_key = "AKIAQIW66DN25R3K2DOW"
  secret_key = "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"
}

resource "aws_instance" "myec2" {
   ami = "ami-082b5a644766e0e6f"
   instance_type = lookup(var.instance_type,terraform.workspace)
}

variable "instance_type" {
  type = "map"

  default = {
    default = "t2.nano"
    dev     = "t2.micro"
    prd     = "t2.large"
  }
}
```

Workspace will be located in terraform.tfstate.d

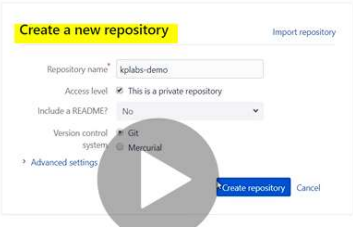
All other folders are root/default workspace

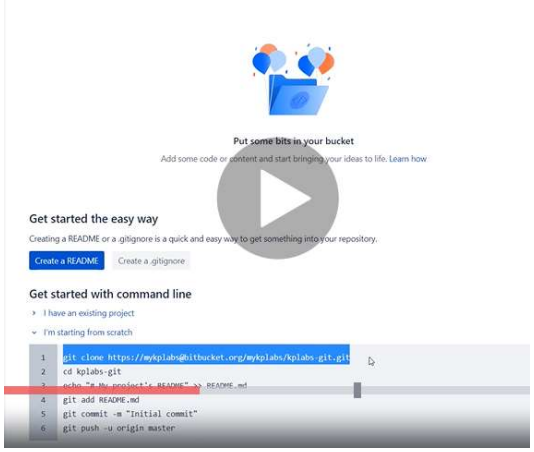


Terraform codes stored in Git repository instead of local machine where other members of team can pull the code to work on it.



Bitbucket.org





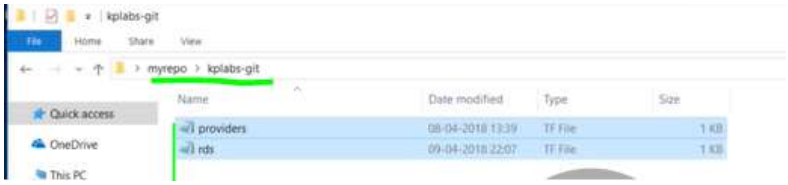
Download git for windows and install git client

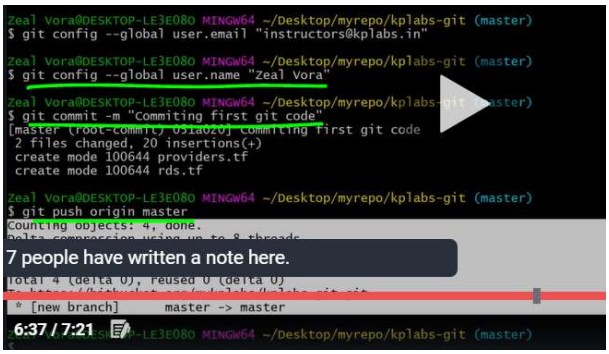
Launch Git Bash

Create new folder in windows

Switch to the same directory and execute the clone command

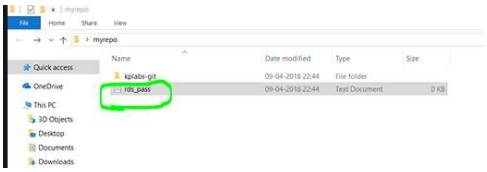
```
MINGW64:/c/Users/Zeal Vora/Desktop/myrepo

Zeal Vora@DESKTOP-LE3E08O MINGW64 ~
$ pwd
/c/Users/Zeal Vora

Zeal Vora@DESKTOP-LE3E08O MINGW64 ~
$ cd Desktop/myrepo/

Zeal Vora@DESKTOP-LE3E08O MINGW64 ~/Desktop/myrepo
$ git clone https://mykplabs@bitbucket.org/mykplabs/kplabs-git.git
Cloning into 'kplabs-git'...
warning: You appear to have cloned an empty repository.

Zeal Vora@DESKTOP-LE3E08O MINGW64 ~/Desktop/myrepo
$
```

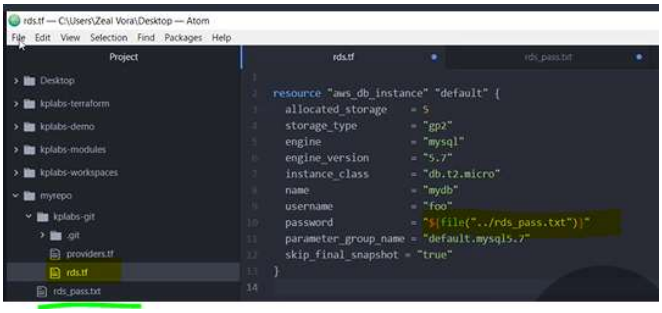




Files are moved to Git from desktop

Create password file outside of git repo

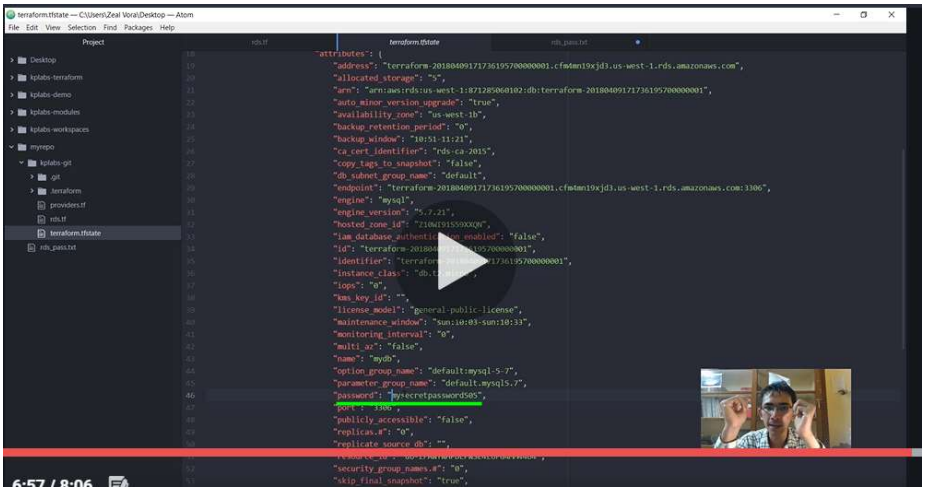




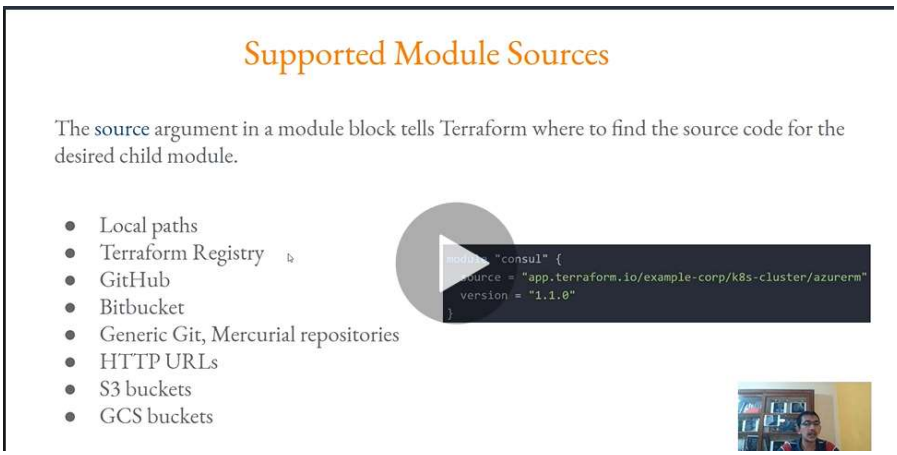
A password file(rds_pass.txt) is created which has password details and in rds.tf file with code details

if the file terraform.tfstate is send to git then password is also stored which is not recommended.

Committing the terraform.tfstate file in git repository is not recommended since it has password details.


terraform.tfstate — C:\Users\Zeal Vora\Desktop — Atom
File Edit View Selection Find Packages Help
Project
Desktop
kplabs-terraform
kplabs-demo
kplabs-modules
kplabs-workspaces
myrepo
kplabs-git
.git
.terraform
providers.tf
rds.tf
terraform.tfstate
rds_pass.txt
rds.tf
terraform.tfstate
rds_pass.txt
"attributes": {
"address": "terraform-20180409171736195700000001.cfmdmn19xjd3.us-west-1.rds.amazonaws.com",
"allocated_storage": "5",
"arn": "arn:aws:rds:us-west-1:871285060102:db:terraform-20180409171736195700000001",
"auto_minor_version_upgrade": "true",
"availability_zone": "us-west-1b",
"backup_retention_period": "0",
"backup_window": "10:51-11:21",
"ca_cert_identifier": "rds-ca-2015",
"copy_tags_to_snapshot": "false",
"db_subnet_group_name": "default",
"endpoint": "terraform-20180409171736195700000001.cfmdmn19xjd3.us-west-1.rds.amazonaws.com:3306",
"engine": "mysql",
"engine_version": "5.7.21",
"hosted_zone_id": "Z1MI91559XXQN",
"iam_database_authentication_enabled": "false",
"id": "terraform-20180409171736195700000001",
"identifier": "terraform-20180409171736195700000001",
"instance_class": "db.t2.micro",
"iops": "0",
"kms_key_id": "",
"license_model": "general-public-license",
"maintenance_window": "sun:10:03-sun:10:33",
"monitoring_interval": "0",
"multi_az": "false",
"name": "mydb",
"option_group_name": "default:mysql-5-7",
"parameter_group_name": "default.mysql5.7",
"password": "myrecretpassword505",
"port": "3306",
"publicly_accessible": "false",
"replicas.#": "0",
"replicate_source_db": "",
"security_group_names.#": "0",
"skip_final_snapshot": "true",
6:57 / 8:06

GIT module


Supported Module Sources
The source argument in a module block tells Terraform where to find the source code for the desired child module.
Local paths
Terraform Registry
GitHub
Bitbucket
Generic Git, Mercurial repositories
HTTP URLs
S3 buckets
GCS buckets
module "consul" {
source = "app.terraform.io/example-corp/k8s-cluster/azurerm"
version = "1.1.0"
}

Git Module Source
Arbitrary Git repositories can be used by prefixing the address with the special git:: prefix.
After this prefix, any valid Git URL can be specified to select one of the protocols supported by Git.
module "vpc" {
source = "git::https://example.com/vpc.git"
}
module "storage" {
source = "git::ssh://username@example.com/storage.git"
}

Referencing to a Branch

By default, Terraform will clone and use the default branch (referenced by HEAD) in the selected repository.

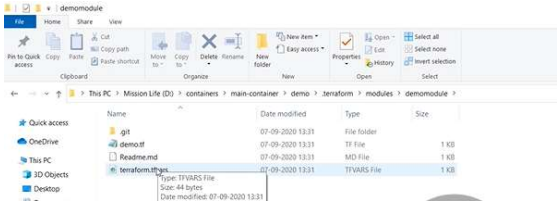
You can override this using the ref argument:

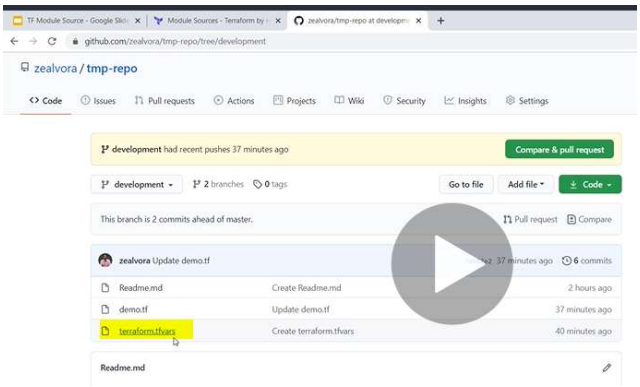
```
module "vpc" {
  source = "git::https://example.com/vpc.git?ref=v1.2.0"
}
```

.

The value of the ref argument can be any reference that would be accepted by the git checkout command, including branch and tag names.

Set the terraform to work with git repo





Terraform and .gitignore

Depending on the environments, it is recommended to avoid committing certain files to GIT

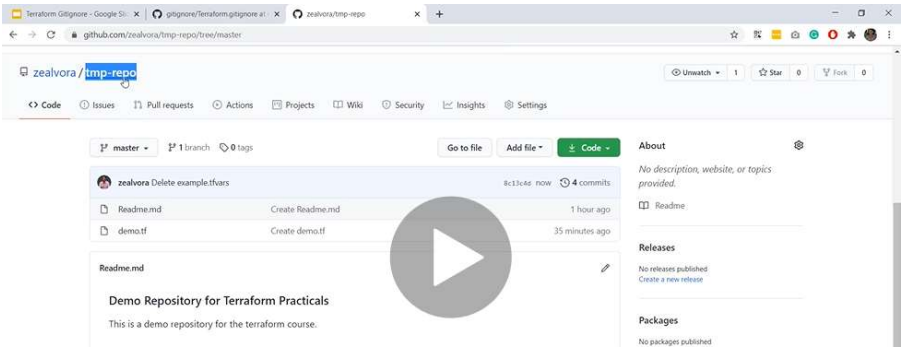
Terraform and .gitignore

Depending on the environments, it is recommended to avoid committing certain files to GIT.

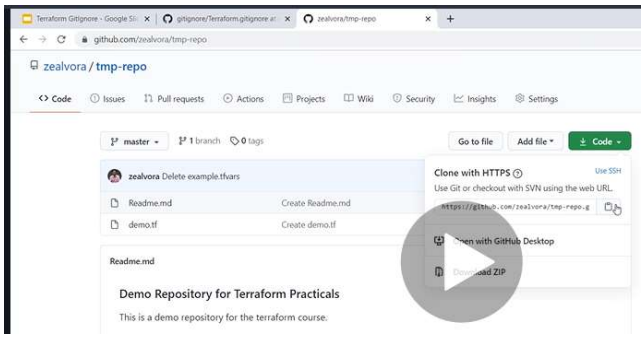
Files to Ignore	Description
.terraform	This file will be recreated when terraform init is run.
terraform.tfvars	Likely to contain sensitive data like usernames/passwords and secrets.
terraform.tfstate	Should be stored in the remote side.
crash.log	If terraform crashes, the logs are stored to a file named crash.log

https://github.com/github/gitignore/blob/master/Terraform.gitignore

Git repo



Clone this repo



Git should be installed in the local system

```
Command Prompt - docker exec -it main bash
bash-4.2# git clone https://github.com/zealvora/tmp-repo.git
Cloning into 'tmp-repo'...
remote: Enumerating objects: 10, done.
remote: Counting objects: 100% (10/10), done.
remote: Compressing objects: 100% (8/8), done.
remote: Total 10 (delta 1), reused 0 (delta 0), pack-reused 0
Unpacking objects: 100% (10/10), done.
bash-4.2#
```

There is demo.tf as expected is cloned with git

```
Select Command Prompt - docker exec -it main bash
bash-4.2# ls -l
total 4
drwxr-xr-x 3 root root 4096 Sep  7 07:10 tmp-repo
bash-4.2# cd tmp-repo/
bash-4.2# ls
Readme.md  demo.tf
bash-4.2#
```

Create example.tfvars with below contents

```
Command Prompt - docker exec -it main bash
  GNU nano 2.9.8                                   example.tfvars

username = "admin"

password = "password
```

```
Command Prompt - docker exec -it main bash
bash-4.2# nano example.tfvars
bash-4.2# terraform apply

An execution plan has been generated and is shown below.
Resource actions are indicated with the following symbols:

Terraform will perform the following actions:

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

Do you want to perform these actions?
  Terraform will perform the actions described above.
  Only 'yes' will be accepted to approve.

  Enter a value: Y
```

As expected it has same contents as git demo.tf

```
Select Command Prompt - docker exec -it main bash
An execution plan has been generated and is shown below.
Resource actions are indicated with the following symbols:

Terraform will perform the following actions:

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

Do you want to perform these actions?
  Terraform will perform the actions described above.
  Only 'yes' will be accepted to approve.

  Enter a value: yes

Apply complete! Resources: 0 added, 0 changed, 0 destroyed.
Outputs:

tes2 = [
  "host1",
  "host2",
  "host3",
]
test = [
  "host1",
  "host3",
```

After terraform apply these files will be visible

```
Select Command Prompt - docker exec -it main bash
bash-4.2# ls -l
total 16
-rw-r--r-- 1 root root  98 Sep  7 07:10 Readme.md
-rw-r--r-- 1 root root 225 Sep  7 07:10 demo.tf
-rw-r--r-- 1 root root  42 Sep  7 07:11 example.tfvars
-rw-r--r-- 1 root root 592 Sep  7 07:11 terraform.tfstate
```

Create .gitignore file to exclude all these files during commit

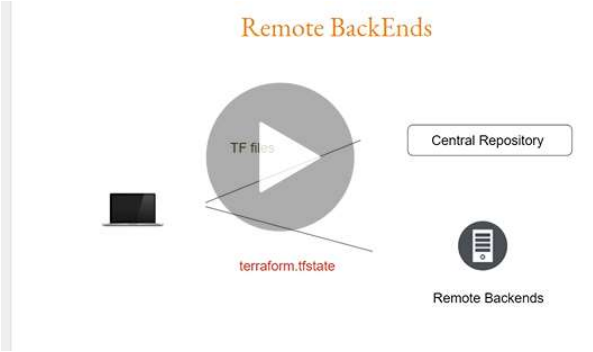
```
Command Prompt - docker exec -it main bash
  GNU nano 2.9.8                                   .gitignore

.terraform
*.tfvars
terraform.tfstate
```

.gitignore file is created as seen below

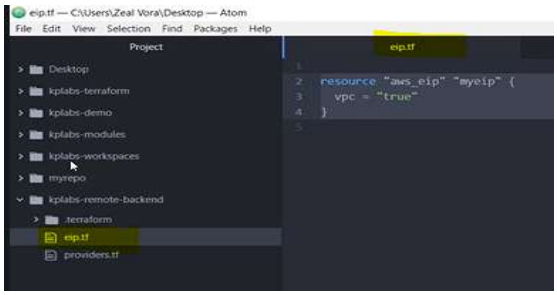
In git status output results show only .gitignore is only included during commit in git repository while rest of the files are excluded as expected.

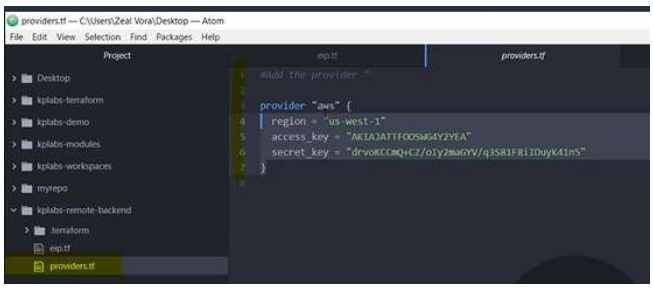
```
Select Command Prompt - docker exec -it main bash
bash-4.2# ls -la
total 36
drwxr-xr-x 4 root root 4096 Sep  7 07:12 .
drwxr-xr-x 3 root root 4096 Sep  7 07:10 ..
drwxr-xr-x 8 root root 4096 Sep  7 07:12 .git
-rw-r--r-- 1 root root   38 Sep  7 07:12 .gitignore
drwxr-xr-x 3 root root 4096 Sep  7 07:11 .terraform
-rw-r--r-- 1 root root   98 Sep  7 07:10 Readme.md
-rw-r--r-- 1 root root  225 Sep  7 07:10 demo.tf
-rw-r--r-- 1 root root   42 Sep  7 07:11 example.tfvars
-rw-r--r-- 1 root root  592 Sep  7 07:11 terraform.tfstate
bash-4.2# git status
On branch master
Your branch is up to date with 'origin/master'.

Untracked files:
  (use "git add <file>..." to include in what will be committed)
        .gitignore

nothing added to commit but untracked files present (use "git add" to track)
bash-4.2#
```

TF code is stored in central git repository and terraform.tfstate files are stored in remote backends such as s3 for everyone in team to collaborate.

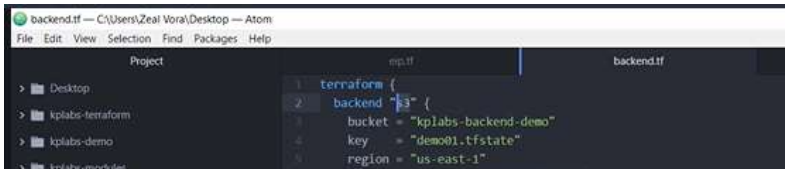


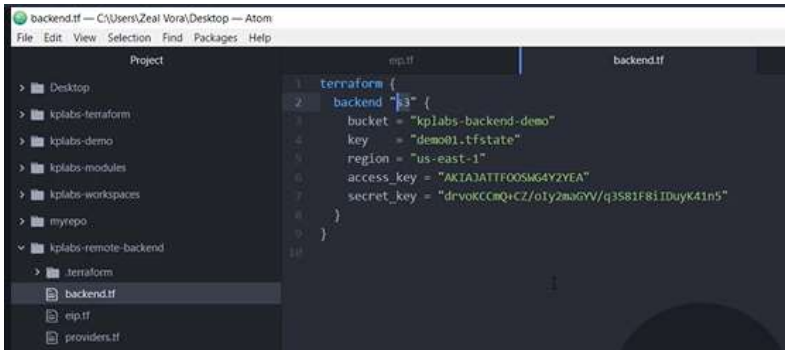
https://www.terraform.io/docs/backends/types/s3.html





Backend.tft file is created which has s3 details to store the terraform.tftstate files are stored.





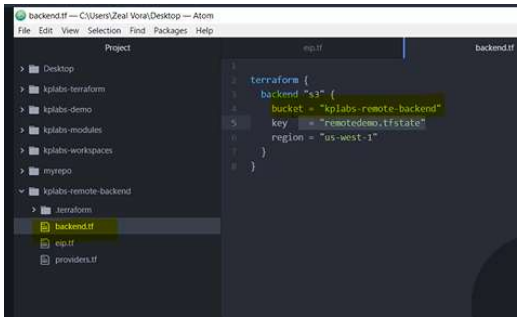
Supported BackEnd Types

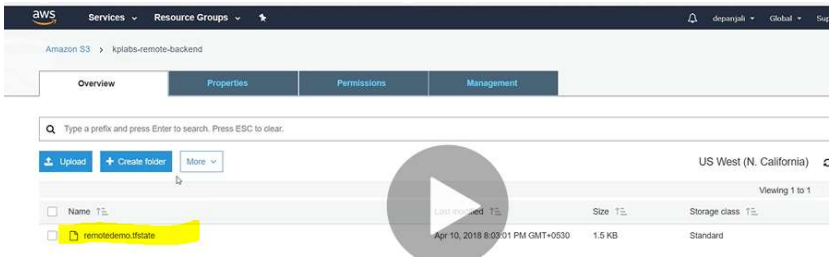
Terraform supports various types of remote backends which can be used to store state data.

As of now, we were storing state data in local and GIT repository.

Depending on remote backends which is being used, there can be various features.

Standard BackEnd Type: State Storage and Locking
Enhanced BackEnd Type: All features of Standard + Remote Management





Understanding State Lock

Whenever you are performing write operation, terraform would lock the state file.

This is very important as otherwise during your ongoing terraform apply operations, if others also try for the same, it would corrupt your state file.

Example:

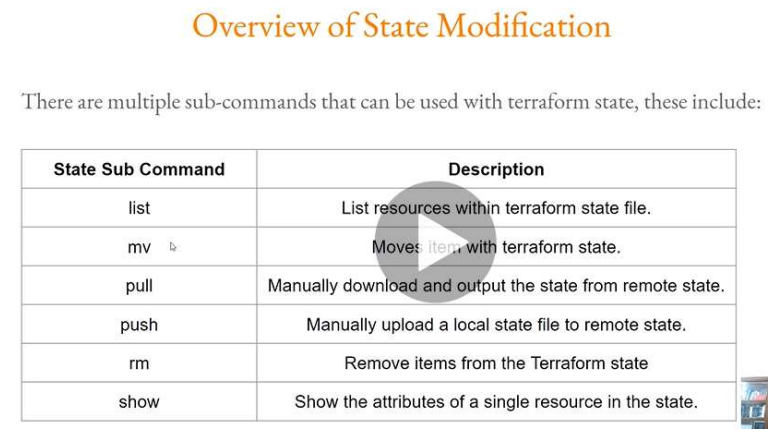
- Person A is terminating the RDS resource which has associated rds.tfstate file
- Person B has now tried resizing the same RDS resource at same time.

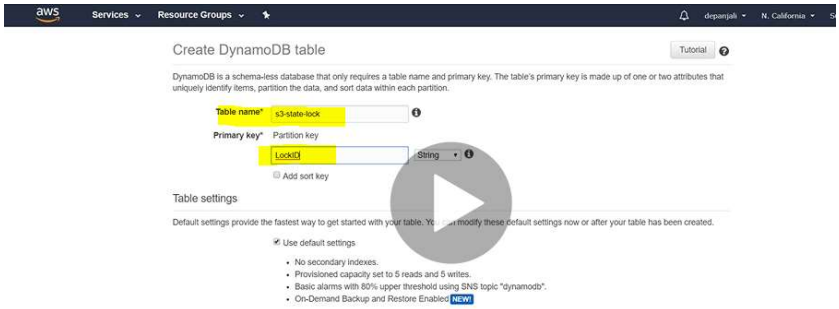
When terraform plan is executed it locks the file but when someone else is also trying to execute terraform plan at same time it will give below error message

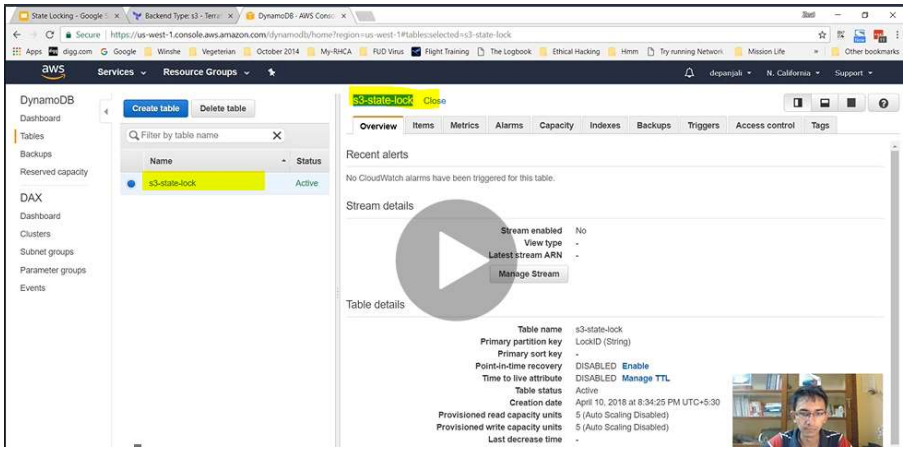
You can implement state locking with remote backends such as S3

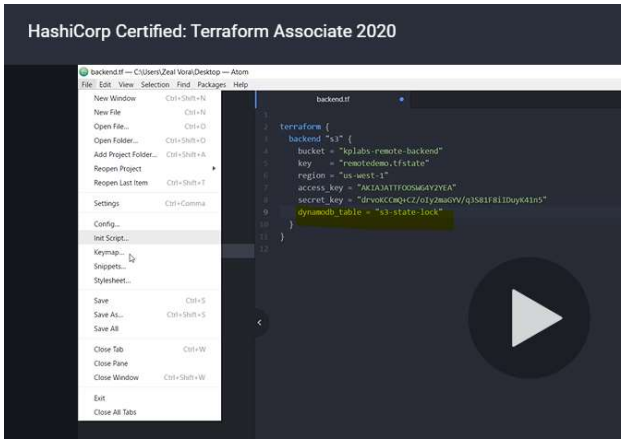
https://www.terraform.io/docs/backends/types/s3.html

Create a dynamodb table and add entry in backend to lock id is generated when simultaneous changes are made.









Overview of State Modification

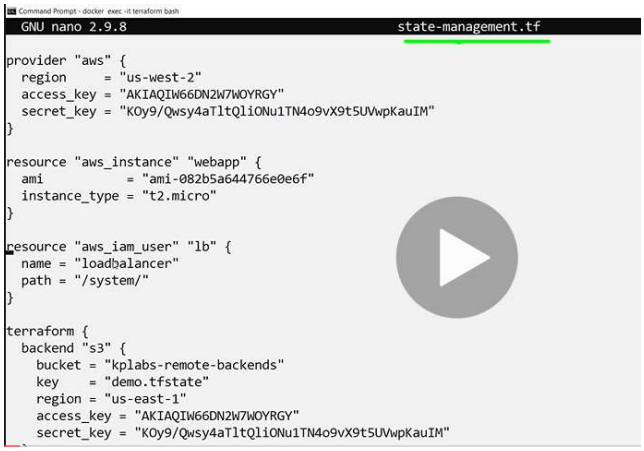
There are multiple sub-commands that can be used with terraform state, these include:

State Sub Command	Description
list	List resources within terraform state file.
mv	Moves items with terraform state.
pull	Manually download and output the state from remote state.
push	Manually upload a local state file to remote state.
rm	Remove items from the Terraform state
show	Show the attributes of a single resource in the state.

Sub Command - List

The terraform state list command is used to list resources within a Terraform state.

```
bash-4.2# terraform state list
aws_iam_user.lb
aws_instance.webapp
```



```
Select Command Prompt - docker exec -it terraform bash
  GNU nano 2.9.8                                    state-management.tf

  name = "loadbalancer"
  path = "/system/"
}

terraform {
  backend "s3" {
    bucket = "kplabs-remote-backends"
    key    = "demo.tfstate"
    region = "us-east-1"
    access_key = "AKIAQIW66DN2W7WOYRGY"
    secret_key = "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"
  }
}
```

State list

```
Select Command Prompt - docker exec -it terraform bash
bash-4.2# terraform state list
aws_iam_user.lb
aws_instance.webapp
bash-4.2#
```

```
Select Command Prompt - docker exec -it terraform bash
  GNU nano 2.9.8                                    state-management.tf

provider "aws" {
  region     = "us-west-2"
  access_key = "AKIAQIW66DN2W7WOYRGY"
  secret_key = "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"
}

resource "aws_instance" "webapp" {
  ami           = "ami-082b5a644766e0e6f"
  instance_type = "t2.micro"
}

resource "aws_iam_user" "lb" {
  name = "loadbalancer"
  path = "/system/"
}

terraform {
  backend "s3" {
    bucket = "kplabs-remote-backends"
    key    = "demo.tfstate"
    region = "us-east-1"
    access_key = "AKIAQIW66DN2W7WOYRGY"
    secret_key = "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"
```

Sub Command - Move

The terraform state mv command is used to move items in a Terraform state.

This command is used in many cases in which you want to rename an existing resource without destroying and recreating it.

Due to the destructive nature of this command, this command will output a backup copy of the state prior to saving any changes

Overall Syntax:

terraform state mv [options] SOURCE DESTINATION

We are changing EC2 instance name from webapp to myec2 while executing terraform plan it try to destroy and then recreate EC2.

After move command if terraform plan is executed there are no changes

```
Select Command Prompt - docker exec -it terraform bash
  GNU nano 2.9.8                                    state-management.tf

provider "aws" {
  region     = "us-west-2"
  access_key = "AKIAQIW66DN2W7WOYRGY"
  secret_key = "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"
}

resource "aws_instance" "webapp" {
  ami           = "ami-082b5a644766e0e6f"
  instance_type = "t2.micro"
}

resource "aws_iam_user" "lb" {
  name = "loadbalancer"
  path = "/system/"
}

terraform {
  backend "s3" {
    bucket = "kplabs-remote-backends"
    key    = "demo.tfstate"
    region = "us-east-1"
    access_key = "AKIAQIW66DN2W7WOYRGY"
    secret_key = "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"
```

```
Command Prompt - docker exec -it terraform bash
  GNU nano 2.9.8                                    state-management

provider "aws" {
  region     = "us-west-2"
  access_key = "AKIAQIW66DN2W7WOYRGY"
  secret_key = "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"
}

resource "aws_instance" "myec2" {
  ami           = "ami-082b5a644766e0e6f"
  instance_type = "t2.micro"
}

resource "aws_iam_user" "lb" {
  name = "loadbalancer"
  path = "/system/"
}

terraform {
  backend "s3" {
    bucket = "kplabs-remote-backends"
    key    = "demo.tfstate"
    region = "us-east-1"
    access_key = "AKIAQIW66DN2W7WOYRGY"
    secret_key = "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"
```

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

```
Command Prompt - docker exec -it terraform bash
bash-4.2# terraform state mv aws_instance.webapp aws_instance.myec2
Move "aws_instance.webapp" to "aws_instance.myec2"
Successfully moved 1 object(s).
bash-4.2#
```

```
Select Command Prompt - docker exec -it terraform bash
bash-4.2# terraform plan
Refreshing Terraform state in-memory prior to plan...
The refreshed state will be used to calculate this plan, but will not be
persisted to local or remote state storage.

aws_iam_user.lb: Refreshing state... [id=loadbalancer]
aws_instance.myec2: Refreshing state... [id=i-0cb034be6dd6e1f8d]

------------------------------------------------------------------------

No changes. Infrastructure is up-to-date.

This means that Terraform did not detect any differences between your
configuration and real physical resources that exist. As a result, no
actions need to be performed.
bash-4.2#
bash-4.2#
```

Sub Command - Pull

The terraform state pull command is used to manually download and output the state from remote state.

This is useful for reading values out of state (potentially pairing this command with something like jq).

U get all information of state

```
Command Prompt - docker exec -it terraform bash
bash-4.2#
bash-4.2# terraform state pull
{
  "version": 4,
  "terraform_version": "0.12.24",
  "serial": 4,
  "lineage": "b5770e9f-ef67-44e4-5904-7741d2cbbd9e",
  "outputs": {},
  "resources": [
    {
      "mode": "managed",
      "type": "aws_iam_user",
      "name": "lb",
      "provider": "provider.aws",
      "instances": [
        {
          "schema_version": 0,
          "attributes": {
            "arn": "arn:aws:iam::018721151861:user/system/loadbalancer",
            "force_destroy": false,
```

Sub Command - Remove

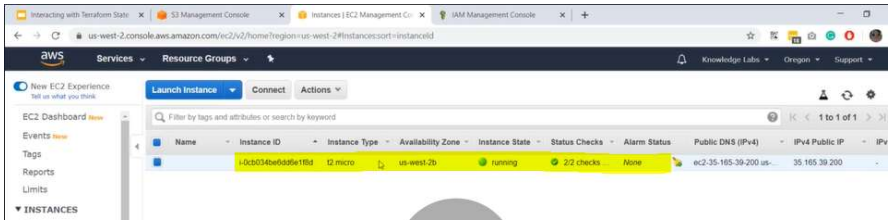
The terraform state rm command is used to remove items from the Terraform state.

Items removed from the Terraform state are not physically destroyed.

Items removed from the Terraform state are only no longer managed by Terraform

For example, if you remove an AWS instance from the state, the AWS instance will continue running, but terraform plan will no longer see that instance.

```
Select Command Prompt - docker exec -it terraform bash
bash-4.2# terraform state list
aws_iam_user.lb
aws_instance.myec2
bash-4.2# terraform state rm aws_instance.myec2
Removed aws_instance.myec2
Successfully removed 1 resource instance(s).
bash-4.2#
```



Sub Command - Show

The terraform state show command is used to show the attributes of a single resource in the Terraform state.

```
bash-4.2# terraform state show aws_instance.webapp
# aws_instance.webapp:
resource "aws_instance" "webapp" {
    ami                          = "ami-082b5a644766e0e6f"
    arn                          = "arn:aws:ec2:us-west-2:018721151861:instance/i-0107ea9ed06c467e0"
    associate_public_ip_address  = true
    availability_zone            = "us-west-2b"
    cpu_core_count               = 1
    cpu_threads_per_core         = 1
    disable_api_termination      = false
    ebs_optimized                = false
    get_password_data            = false
    id                           = "i-0107ea9ed06c467e0"
    instance_state               = "running"
    instance_type                = "t2.micro"
```

```
bash-4.2# terraform state show aws_iam_user.lb
# aws_iam_user.lb:
resource "aws_iam_user" "lb" {
    arn           = "arn:aws:iam::018721151861:user/system/loadbalancer"
    force_destroy = false
    id            = "loadbalancer"
    name          = "loadbalancer"
    path          = "/system/"
    tags          = {}
    unique_id     = "AIDAQIW66DN2W64TNBEER"
}
bash-4.2#
```

EC2 instance created manually that we are trying to import to Terraform

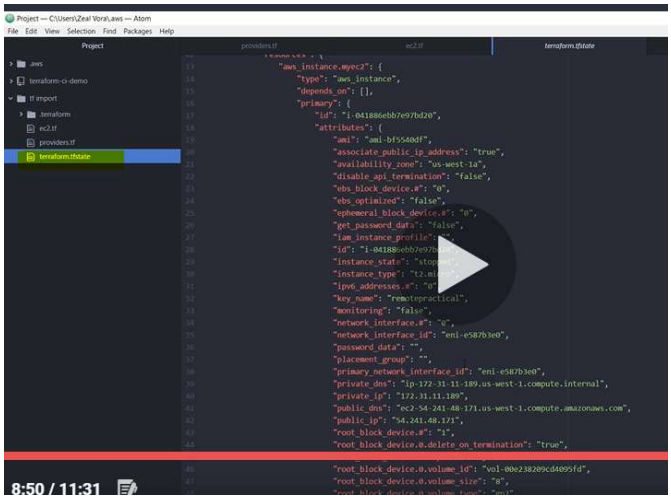
Create Terraform file and mention all properties of Manual instance in the code

```
resource "aws_instance" "myec2" {
  ami = "ami-bf5540df"
  instance_type = "t2.micro"
  vpc_security_group_ids = ["sg-6ae7d613"]
  key_name = "remotepractical"
  subnet_id = "subnet-9e3cfbc5"

  tags {
    Name = "manual"
  }
}
```

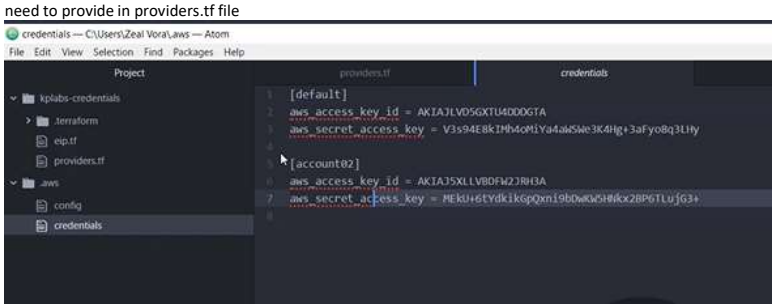
```
C:\Users\Zeal Vora\Desktop\tf import>terraform init

Initializing provider plugins...
- Checking for available provider plugins on https://releases.hashicorp.com...
```

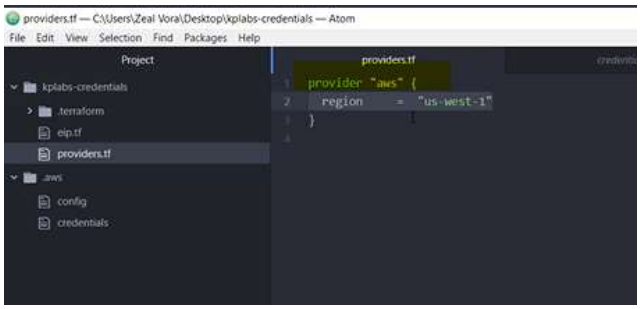
```
C:\Users\Zeal Vora\Desktop\tf import>terraform import aws_instance.myec2 i-041886ebb7e97bd20
aws_instance.myec2: Importing from ID "i-041886ebb7e97bd20"...
aws_instance.myec2: Import complete!
  Imported aws_instance (ID: i-041886ebb7e97bd20)
aws_instance.myec2: Refreshing state... (ID: i-041886ebb7e97bd20)

Import successful!

The resources that were imported are shown above. These resources are now in
your Terraform state and will henceforth be managed by Terraform.
```

It is included in terraform.tfsate file



If AWS Cli is configured and terraform take credentials from credentials files so there is no need to provide in providers.tf file

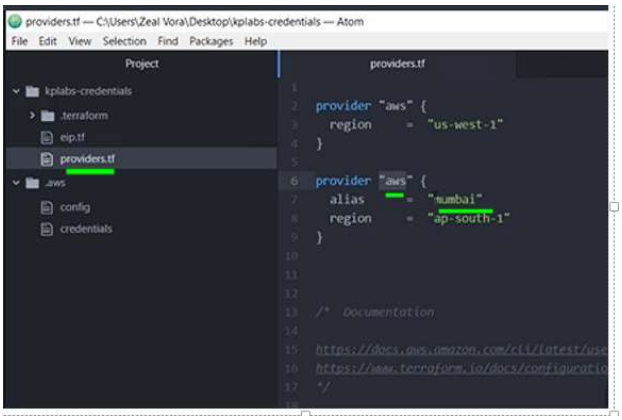


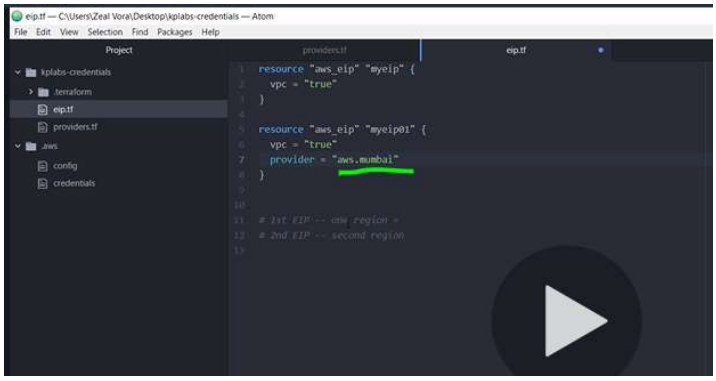


Single Provider Multiple Configuration

Till now, we have been hardcoding the aws-region parameter within the providers.tf

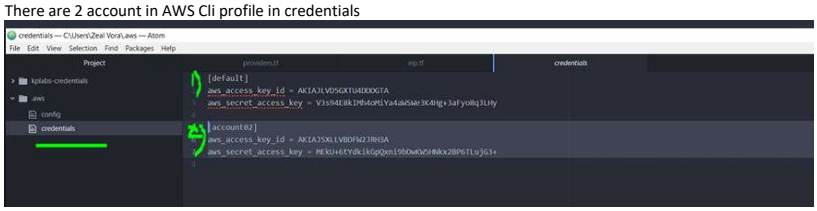
This means that resources would be created in the region specified in the providers.tf file.

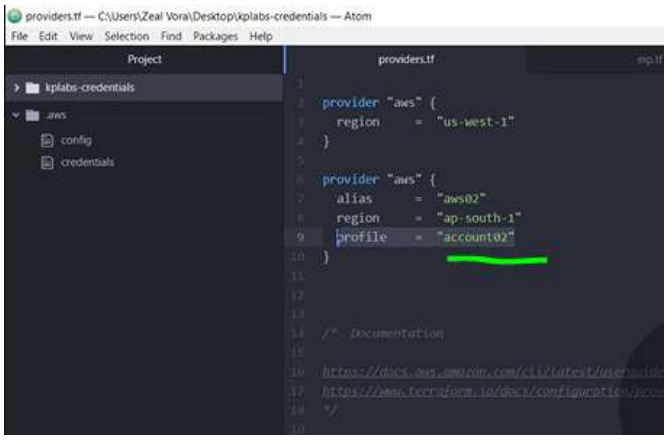


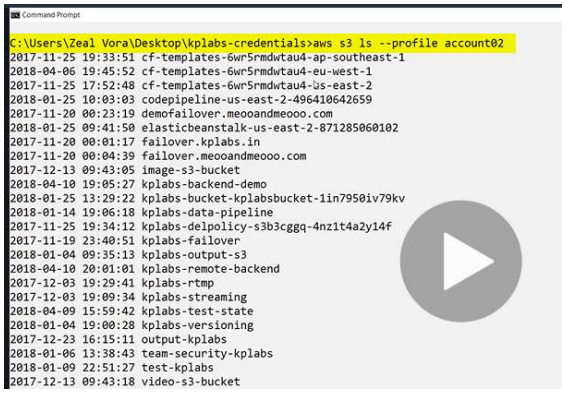


Single Provider Multiple Configuration

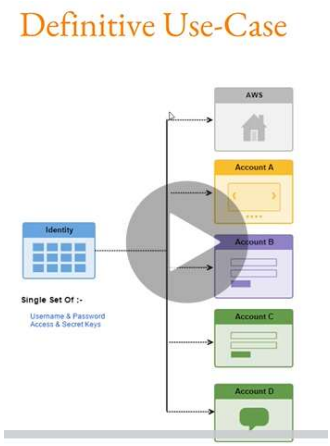
There are 2 account in AWS Cli profile in credentials



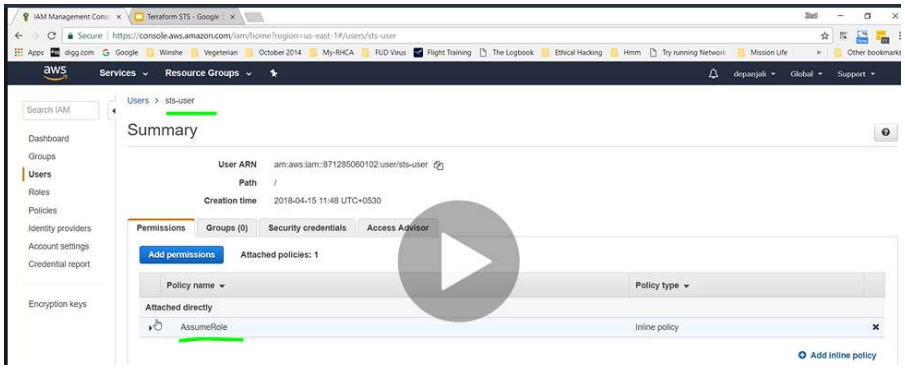


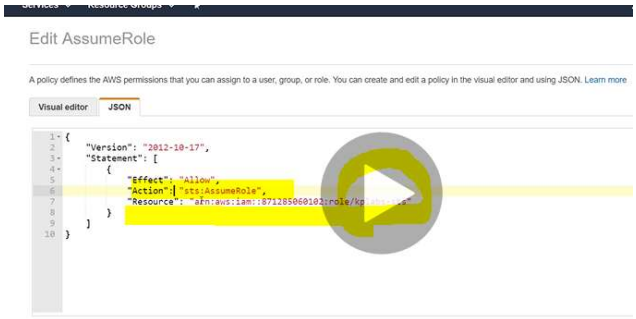


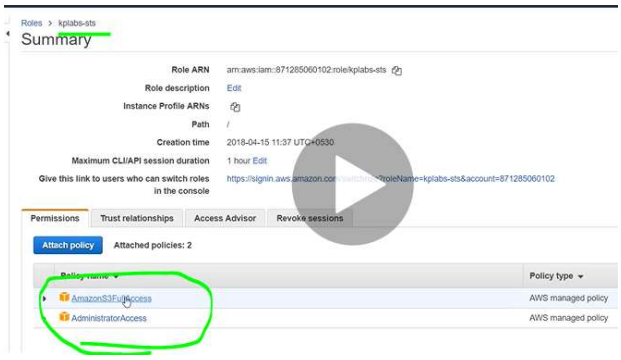
AWS assume role

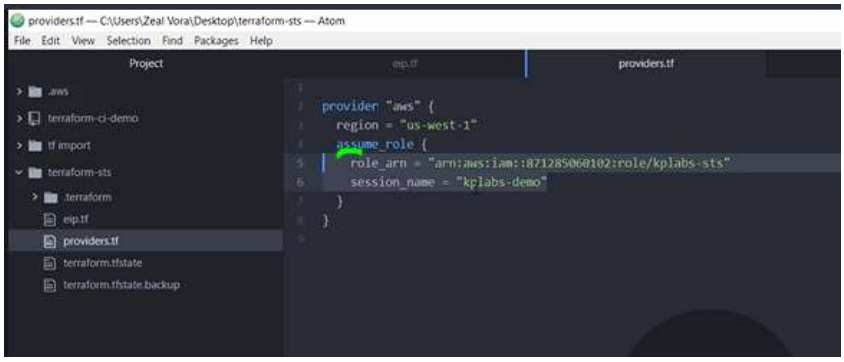


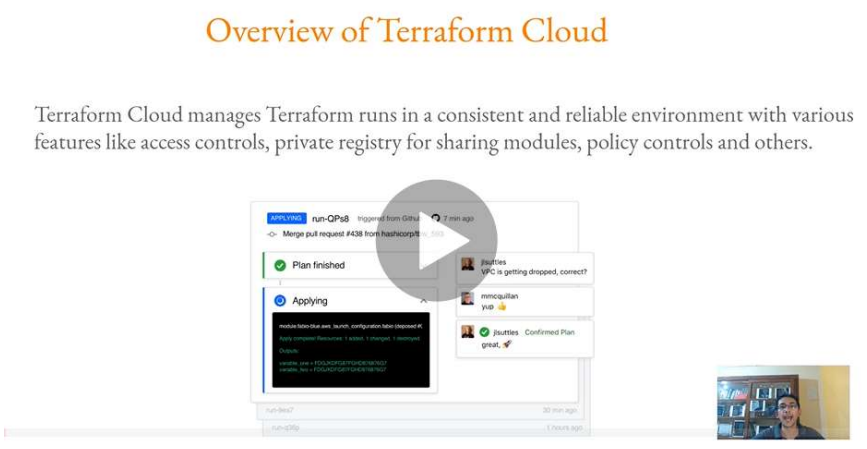
Here is the sts-user which has assume role of kplabs-sts which admin access and s3 full access











An Introduction to Terraform Templates

A look at Terraform templates, providers, resources, and modules

NOV 12, 2020 | ABHISHEK HUGAR

Welcome back to our blog series on Terraform, where we break down the various components piece-by piece, for you to become an expert. In our current series, we're going to discuss everything you need to know about resources, providers, and modules. Note: This blog is geared towards those who are familiar with the basics of Terraform. If you're new to Infrastructure as Code, we suggest that you first read our previous post on "Terraform Tutorial Guides" to build a strong foundation before diving into the current content.

We believe that in order to understand Terraform templates, there are a number of subcategories to also understand. That's why we included providers, resources, and modules in our explanation.
In this article, we'll give you concise and actionable explanations to:

- Terraform Templates
- Defining Providers
- Declaring Resources
- An Overview of Modules

An Overview of Terraform Templates

A terraform template is a collection of files that, together, define the state of your infrastructure to be achieved. They include different configuration files such as variables, resources, and modules.

You can keep the terraform template files separately, or all under one configuration file--it's usually your own choice. Before you begin building any terraform templates, here are a few key terms and concepts to familiarize yourself with:

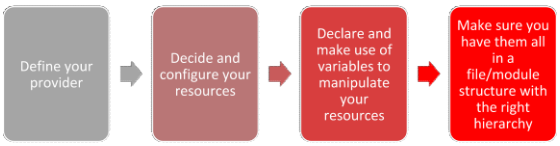
- **Blocks:** These act like containers, and always begin with the name of the type of block, which can be a resource, variable, or provider. Depending on the type of block, the number of labels will be defined. The content is then defined within opening and closing curly brackets ('{' and '}').
- **Comments:** To increase the readability of the code, you may always include comments to specify the function of any line of code. You may either use "#" or "//" to define the single line comments. Multi-line comments are written between "/*" and "*/".
- **Arguments:** These are the name and value pairs inside a block, where a value is assigned to a particular name. The name is called an "identifier," which is followed by the equal to (=) sign and then a particular value or an expression is written on the right-hand side of the "=" sign. An identifier or name may contain a letter, a number, an underscore (_) or a dash (-), and should never start with a number.
- **Functions**: There is support for pre-defined functions, which may be used within expressions in order to manipulate values. You can use functions by using function names, followed by writing values that need to be passed to the function between the opening and closing brackets ('(' and ')' respectively). However, you cannot define your own functions and are only able to use built-in ones. While it's not possible to cover all of the built-in functions in this blog post, we encourage you to take time and study the pre-built ones provided by Terraform.

Terraform templates are always in a modular structure. In simpler terms, we can say that they're set up in a file hierarchy, where each configuration file contains a name with the extension ".tf" (or in JSON format, ".tf.json"). Each configuration file needs to be UTF-8 encoded and is written as a block.
The below image indicates the general syntax of how we may write blocks:

```
<BLOCK TYPE> "<BLOCK LABEL>" "<BLOCK LABEL>" {
  # Block body
  <IDENTIFIER> = <EXPRESSION> # Argument
}
```

Source: **Configuration Language**

Let's take a look at the steps necessary to set up your configuration:



We've already covered the topic of defining and writing variable files in our previous blog, and will dive into the other steps below.

Defining Providers

When we say providers, we're actually talking about the cloud or self-hosted service providers who enable all of your requirements based on your configuration files. When it comes to Terraform, providers are available as installable plugins, where each plugin is available in different versions and each version will offer its own set of resources, arguments that are accepted, and attributes that can be exported.
These installable plugins are automatically installed when you initialize your working directory, or in other words, when you run the "init" command. If you already have a working directory that has been initialized/applied, and you make changes to add a new provider, the new provider is automatically installed when you initialize the directory again. But, how can we specify which provider should be selected?

This specification is done under the "required_providers" block. This particular block is written inside the "terraform" block. Like all other blocks that we've seen, these blocks also start and end with an opening and closing curly brackets. The "required_providers" block contains a local name for the block, the URL or the address of the source of the provider, and an optional value specifying which version should be downloaded.
Let's see an example of this:

```
terraform {
  required_providers {
    mycloud = {
      source  = "mycorp/mycloud"
      version = "~> 1.0"
    }
  }
}
```

Source: **Requirements Block**
NOTE: Only 1 provider can be specified for each block.
A few providers will also need you to write configuration blocks before they're used. These blocks will usually contain details such as the region and URLs. These can be written inside the "provider" block. It begins with the keyword "provider," followed by the local name defined in the "required_providers" and between the curly brackets, you may write arguments that are always provider-specific.
For example:

```
provider "google" {
  project = "acme-app"
  region  = "us-central1"
}
```

Source: **Writing Configurations**

NOTE: You may reference only those values that have been defined before the configuration, such as input variables. This means that the attributes exported by the resources cannot be used in the expressions.
Providers have meta-arguments that may be used:

ALIAS

- This is used when you require multiple configurations for the same provider. You may define multiple "provider" configuration blocks with the same name.
- From the second block with the same name onwards, use the "alias" meta-argument to provide an alternate name to the block.

VERSION

- Determines the version of the provider plugin, just like "required_provider".
- Hashicorp suggests not using this meta-argument in the "providers" block and instead use it in the "required_providers" itself.

Declaring Resources

To understand resources, you must first question, "What exactly are resources when it comes to Infrastructure?". Well, they could be instances, storage space, DNS records, virtual networks, etc. So, you see now that resources are the basic requirements in your infrastructure. Terraform allows you to write blocks in your configuration to define all the resources that should be part of your infrastructure.
Resource blocks permit you to write configurations for the type of infrastructure object you will need. If it's a new configuration that is being defined, then it is created on the "apply" command and the necessary "state file" is created. If it's about updating a current object, then the state file is compared with the new object and necessary changes are updated.
The logical next step is to know how to write a resource block for a configuration.
Every block in this particular situation starts with the keyword "resource" and is followed by the type and then by a locally defined name. We then write the requirement configurations within the opening and closing curly brackets, "{" and "}" respectively. Naming conventions specify that the resource name should begin with a letter or underscore (_) and contain only letters, digits, underscores and dashes.
Example:

```
resource "aws_instance" "web" {
  ami           = "ami-a1b2c3d4"
  instance_type = "t2.micro"
}
```

Source: **Resource Syntax**
The "type" defined in the block is always the cloud provider that we'll use to implement our infrastructure. Each provider will have their own types of configurations that can be used in the block between the curly brackets. We'll talk more about the nuances of providers in a bit. However, at any stage, feel free to browse and take a look at the available providers and their documentation.
If you're looking to access information regarding the current configuration in the same module, use the syntax below:

<TYPE_OF_RESOURCE>.<RESOURCE_NAME>.<ATTRIBUTE>
If you take into consideration the example block we've shown above, then to access an attribute "ami" in that block, you have to write a line as "aws_instance.web.ami".
The following image shows a list of meta-arguments and their basic uses:

DEPENDS_ON	• Specifies dependency & order of execution • Only used if there are hidden dependencies Terraform cannot conclude on its own
COUNT	• Determines number of times to create same resource • Use if resources are similar & arguments don't involve strings
FOR_EACH	• Same usage as count, but preferred if your arguments involve strings or maps
PROVIDER	• Automatically dettermined based on resource type • Can override and specify your own provider
LIFECYCLE	• Every resource has lifecycle that begins when created & ends when destroyed • Alter with: create_before_destroy, prevent_destroy, ignore_changes

To keep this guide concise, we've decided to simply provide an overview of meta-arguments. If you're looking to go more in-depth, you can find more resources here.

An Overview of Modules

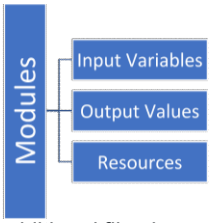
When you imagine Infrastructure as Code, visualize all of the resources and variables in a written and readable format, inside of a particular container or blocks of lines of code. This particular container is what we call "modules." If you're already familiar with programming languages, then simply think of modules as playing the role of a function.

They hold all of the similar resources and variables required for your infrastructure configuration. Similar to functions, one module may call another and return values. As with any other programming languages and functions within it, even in Terraform, it's mandatory to have at least 1 module called the "root module". All of the ".tf" extension files under the working directory constitute the root module.

Reusability is another added advantage, where one module can be defined once and be called in the same configuration or even from other configurations.

You can create a new module by simply creating a new directory and placing your ".tf" configuration files inside them. These files can be accessed by Terraform either by using the local file path or by directly from remote storage, depending on where you have your files.

The below diagram shows the most commonly defined features/files inside a module:



You can include additional files in your directory:

- **README:** A basic description of what the module does, where it can be used, and whether it nests any other modules. It can also be a markdown (.md) file
- **LICENSE:** Indicates which license the module belongs to. It's important because most organizations prefer licensing before usage.
- **MODULES:** If your module has child modules, nest it in the same directory.
- **EXAMPLES:** A few examples to show the usage of the modules. It's also good to have a README file, along with the examples. Preferably place these under the root module.

To define your Terraform template, use this example file structure:

```
.
├── README.md
├── main.tf
├── variables.tf
├── outputs.tf
├── ...
├── modules/
│   ├── nestedA/
│   │   ├── README.md
│   │   ├── variables.tf
│   │   ├── main.tf
│   │   ├── outputs.tf
│   ├── nestedB/
│   ├── .../
├── examples/
│   ├── exampleA/
│   │   ├── main.tf
│   ├── exampleB/
│   ├── .../
```

Source: **Standard Structures**
When a child module is called from the parent in this scenario, it means that you can include the child configurations within the parent, and assign values for input variables. It's vital to remember that you need to include the "source" argument, where the value is the path of the callee.
Example:

```
module "servers" {
  source = "./app-cluster"

  servers = 5
}
```

Source: **Configurations**

NOTE: For any action done on a module, which may be any addition, removal or modification, remember to initialize the working directory using the "init" command, and then apply the changes.
Although the calling module can't access the attributes of the callee, we can declare output values for those arguments which are needed by the calling module. You can do this by assigning the output variable to a local variable, and using the keyword "module": module.child_module_name.argument_name.
NOTE: There are already predefined modules as per the providers in the Terraform Registry, which you may call as you need.
Here are different meta-arguments available for modules and their uses:

VERSION	• Specify which version of module should be called and installed • Applicable only when using modules from "terraform registry"
COUNT AND FOR_EACH	• Determines number of times to create instance for module • For modules, usage of count & for_each is the same with the same data type constraint
PROVIDER	• Provider configurations that need to be passed from the calling module to the callee module is defined • If not definition is given, child module inherits all provider configurations from the parent module
DEPENDS_ON	• Dependency of one module on another & order of their execution is defined here

As you can see, there are many tips and tricks when it comes to Terraform templates, modules, and resources. We hope that you found the above guide useful, and are continuing to learn and grow on your Terraform journey. If you're looking for more support or resources, join our InfraCode Slack to gain valuable help from our community of experts and join the discussion about Infrastructure as Code.

From <https://www.infrastructurecode.io/blog/an-introduction-to-terraform-templates>

Complete Guide to Terraform Templates and Tfvars

Raphael Socher
Nov 20 · 10 min read

We're continuing our beginner series on Terraform tutorials. Note: This content is targeted at newcomers, but can also be a helpful review of core concepts.

Whether you're considering or already using Terraform, at InfraCode, we're firm believers in Hashicorp's vision, but think that it's a heavy lift for small and medium businesses. And, as an individual learner, we understand that the sheer number of unorganized resources make the learning curve steep for newcomers. That's why we decided to put together easy-to-digest guides to make the learning process that much easier.

In this blog, we've compiled overviews of valuable topics for Terraform learners such as:

- The World of Infrastructure on Cloud
- Terraform Templates
- Tfvars (Terraform Variables)
- The Importance of Tfvars and Key Points
- The Terraform State File and Key Points

If you're looking to connect with other Terraform learners or experts, we encourage you to join our Slack community to meet others and exchange valuable ideas.

The World of Infrastructure on Cloud

When discussing setting up an IT enterprise, a major component of the conversation involves setting up the infrastructure. When we say "infrastructure" we're talking about the enterprise's entire collection of hardware, software, networks, data centers, and other facilities that are used to develop, test, operate, monitor, manage and support IT services. When managing the infrastructure through a descriptive model or through code, the same environment is generated every time that code is applied — thereby creating a very fitting name: "Infrastructure as Code (IaC)."

The facilities discussed above can be provided by a cloud service provider, which is referred to as an area known as "Cloud Computing." Cloud computing can be widely classified into 3 services: Infrastructure as a Service (IaaS), Platform as a Service (PaaS), and Software as a Service (SaaS).

IaC can be deployed on the cloud by either pushing the configuration to the designated system or by pulling the required configuration on to the system from the server.

Source: Cloud Computing

Terraform is an open-source tool for building, changing, and versioning infrastructure safely and efficiently. Terraform can manage existing and popular service providers as well as custom in-house solutions. As mentioned above, there are two ways of provisioning your architecture on the cloud, and of these, Terraform follows the "push" process.

Terraform Templates

Configurations on Terraform are usually written within files and then compiled into a folder. A "Terraform template" consists of these files. Each of these folders can be considered as a "module", and every Terraform template contains at least 1 root module, which consists of the following files:

- main.tf: This is where the execution of your Terraform template begins
- variables.tf: A file where you declare all your "input variables"

- outputs.tf: The file with the definitions for every "output value"

Each of these files is written in a block structure, which contains the type of block, its name, and then arguments with assigned values in the form of expressions.
Let's see an example for a block:

In the above image, the text after the symbol "#" is a comment, "resource" is the type of block, and "aws_vpc" and "main" are the labels or names for the block. "cidr_block = var.main_cidr" and "enable_dns_support = true" are the expressions where cidr_block and enable_dns_support are the arguments to which values are assigned after the "=" symbol.
If you found the above paragraph confusing, let's simplify it. We can say that a Terraform template is built like this:

A common question that we hear is "Why is the Terraform template important? What should I focus on?"
Let's put it this way: a Terraform template is the main root on which your entire configuration is dependent. Without it, there is no configuration and no infrastructure.
An even more pressing question is what should you consider in order to design an efficient template?
Look at the image above, where you can see that arguments build-up to blocks, which are then built up as modules, and finally, the collection of these form your final Terraform template.
While this cycle may be obvious, have you considered that when you sit down to design and write the template, you're actually going to move through the cycle in *reverse* order. Therefore, there are some Terraform best practices you should follow when designing:

1. **Understand your infrastructure needs:** This is a crucial step to consider. Irrespective of the size of your enterprise, understanding what exactly is needed as a part of your infrastructure is very important. Based on how well you can analyze and break down your needs, you can build the perfect infrastructure. There are various considerations to think about, such as the resources that are required, your budget, the availability of man-power to code and test the build, etc.
2. **Concentrate on your provider and resources blocks:** Once you have a clear idea of your needs, selecting the right provider should be your priority. We've already emphasized the need for this in our "Best Practices" blog. When it comes to the Terraform template, it's important that you understand how to write your provider and resources block. Always have the exact arguments as per your block, and do not define anything that is not needed and might lead to confusion.
3. **Define your Terraform Variables file:** The tfvars file helps the manipulation of your resources and ensures the exact configurations. Declare all the necessary variables and use them efficiently.
4. **Modular Structure:** Your modules, if structured correctly, move you towards the success of your configuration. Concentrate on small details such as defining the child module inside the parent module and having the correct tfvars file with the corresponding module. This will ensure that you do not face any bugs with your Terraform template file.
5. **Plan before you Apply:** Always keep this step in mind! By using the "plan" command before "apply," you'll be aware of any unforeseen errors or unexpected resource creation/modification.
6. **Remember: Your template is your documentation:** Writing

efficient templates will mean lesser documentation on your end. If your code is written in a simple and readable format, then anyone looking at the code won't require other documentation to understand. However, we also recommend creating a README.md file, as well.

Terraform Variables (Tfvars)

You may have noticed that in the discussion about templates, we repeatedly mentioned tfvars. For a more in-depth discussion about them, check out our recent blog post which goes deeper. They're vital in our configuration process.

Tfvars, just like the variables of any other programming language, are named memory space whose location is known and whose value may be read, written, and re-used. However, in this case, it's not necessary to directly declare the tfvars in the program source. You can define a separate file, every module, and declare the tfvars that are required. You'll notice that these files have the ".tfvars" or ".tfvars.json." extension.

We have three types of tfvars, as depicted in the image below:

The input tfvars can be passed from a parent to child module. Output values are the values that can be returned from a child module to the parent module and can also be printed on the CLI. The output block is almost the same as the variable block, except that the type of block is written with the keyword "output." Local values are those variables that are written within the "locals" block and the arguments defined inside them are restricted to the module inside which they are defined.

We previously mentioned that all files are written in a block structure. The tfvars block structure for input variables can be seen in the below example:

Importance of Tfvars and Key Points

Let's imagine a configuration without variables. You have your provider file with the block defining the provider. There's also the resource block in the resource file, defining the resources that need to be allocated. Is that really enough? No. Let's say, you need storage space, part of the network space or to include the authentication or SSH keys. Each of these needs should be declared within tfvars. Variables are a major part of the configuration and are the means of manipulating your resources to give you the exact infrastructure you need. There is no way to set up your requirements without them. So, learn them well and use them wisely.

For someone just starting out with these concepts, it's always good to keep the following points in mind:

1. **Declaration and Usage:** When you declare variables in the "tfvars" file, it's just the declaration of the variable and not the place where it's in use. Never forget this, or else you may end up not using the tfvars, and resulting in faulty configuration.
2. **Practice Custom Validation:** Though custom validation is a new feature and to some extent can be considered experimental, we would recommend that you start practicing them in your configurations. They enhance your code and can notify you even if there's a small deviation.
3. **Usage of the right data type is healthy:** Terraform doesn't force you to declare the data type of a variable. You can choose not to or use the data type "any." However, we recommend declaring tfvars with their data types, and leaving a description for each. This ensures that in the future, anyone who is working with the data doesn't mess it up. Also,

remember to have your SSH keys as a “map” type, and not a “string” type, so you can declare them under one variable, and reference them anywhere in your configuration using the same tfvars.

The Terraform State File and Key Points

There are many providers available and with the resources that are provided. There’s also a high possibility that more than one person in your team will handle the infrastructure architecture. Every single time a requirement comes up, and someone makes a change to the configuration file, the changes are applied. But, how does Terraform know what to change? You can thank the Terraform state file. For more information about the importance of the Terraform state file and tips on using it, we’ve written a blog post dedicated to the topic. The main responsibility of the Terraform state file is to store the map and store the link between your resource and the remote object that’s created. Every time the resource is changed, the Terraform state file is updated. Every time the resource is destroyed, it’s also tracked. In fact, it even maps the dependency of your modules and resources, to ensure that no object is destroyed outside of its dependency.

When it comes to “Terraform State” always remember the following points:

1. **Avoid manual editing and managing your Terraform state file:** It’s already being managed and updated for you automatically. If the provision is already there for you, why bother manually trying to change the file and potentially undoing your own or a team member’s work? Let the Terraform state file be. Don’t mingle with it.
2. **State Locking:** When your state file is stored on the remote backend, a provision called “State Locking” is provided. This happens automatically on any command that may alter your state and it prevents others from updating the state at the same time and corrupting it. When you are performing a task that is updating the state, you become the owner of the lock, and others cannot acquire it. If the lock fails to be acquired, then the update can’t be performed. Although there is a provision to disable this and even to force unlock, it’s highly recommended that you don’t perform either. If performed, there may be scenarios with multiple writers of the file and a corrupted state file with severe consequences.
3. **Remote file storage:** Do you have multiple team members working on your infrastructure? In this case, we always suggest that you store your Terraform State file on a remote storage space such as cloud service providers. When you store remotely, anytime a team member changes the configuration, the state file is updated. Another advantage of storing files remotely is security. If your configuration involves any credentials or other form of sensitive data, it’s suggested that you store your file on the cloud, so that it’s encrypted and saved instead of saved as plain text on your local system.

- to ensure that the provider version matches the version of Terraform you are using
- providers are released on a separate schedule from Terraform itself; therefore a newer version could introduce breaking changes
(Correct)
- to remove older versions of the provider

Explanation

Providers are plugins released on a separate rhythm from Terraform itself, and so they have their own version numbers. For production use, you should constrain the acceptable provider version via configuration. This helps to ensure that new versions with potentially breaking changes will not be automatically installed by `terraform init` in the future.

Question 9: **Correct**
Which of the following is considered a Terraform plugin?

- Terraform language
- Terraform logic
- Terraform tooling
- Terraform provider
(Correct)

Explanation

Terraform is built on a plugin-based architecture. All providers and provisioners that are used in Terraform configurations are plugins, even the core types such as AWS and Heroku. Users of Terraform are able to write new plugins in order to support new functionality in Terraform.

Question 10: **Correct**
What happens when a `terraform plan` is executed?

- reconciles the state Terraform knows about with the real-world infrastructure
- applies the changes required in the target infrastructure in order to reach the desired configuration
- the backend is initialized and the working directory is prepped
- creates an execution plan and determines what changes are required to achieve the desired state in the configuration files.

(Correct)

Explanation

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.

After a plan has been run, it can be executed by running a `terraform apply`

Question 11: **Correct**
Which of the following best describes the default local backend?

- The local backend is where Terraform Enterprise stores logs to be processed by an log collector
- The local backend stores state on the local filesystem, locks the state using system APIs, and performs operations locally.
(Correct)
- The local backend is the directory where resources deployed by Terraform have direct access to in order to update their current state
- The local backend is how Terraform connects to public cloud services, such as AWS, Azure, or GCP.

Explanation

Information on the default local backend can be found at this link.

Example:

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

Question 12: **Correct**
Select the answer below that completes the following statement:

Terraform Cloud can be managed from the CLI but requires _____?

- a username and password
- a TOTP token
- authentication using MFA
- an API token
(Correct)

Explanation

API and CLI access are managed with API tokens, which can be generated in the Terraform Cloud UI. Each user can generate any number of personal API tokens, which allow access with their own identity and permissions. Organizations and teams can also generate tokens for automating tasks that aren't tied to an individual user.

Question 13: **Correct**
Which of the following connection types are supported by the `remote-exec` provisioner? (select two)

- winrm
(Correct)
- ssh
(Correct)
- rdp
-

smb

Explanation

The remote-exec provisioner invokes a script on a remote resource after it is created. The remote-exec provisioner supports both ssh and winrm type connections.

More information on remote-exec can be found at this link.

Question 14: **Correct**
HashiCorp offers multiple versions of Terraform, including Terraform open-source, Terraform Cloud, and Terraform Enterprise. Which of the following Terraform features are only available in the Enterprise edition? (select four)

- Audit Logs
(Correct)
- Private Module Registry
- Private Network Connectivity
(Correct)
- Clustering
(Correct)
- Sentinel
- SAML/SSO
(Correct)

Explanation

While there are a ton of features that are available to open source users, many features that are part of the Enterprise offering are geared towards larger teams and enterprise functionality. To see what specific features are part of Terraform Cloud and Terraform Enterprise, check out this link.

Question 15: **Correct**
In regards to deploying resources in multi-cloud environments, what are some of the benefits of using Terraform rather than a provider's native tooling? (select three)

- Terraform can help businesses deploy applications on multiple clouds and on-premises infrastructure
(Correct)
- Terraform can manage cross-cloud dependencies
(Correct)
- Terraform simplifies management and orchestration, helping operators build large-scale, multi-cloud infrastructure
(Correct)
- Terraform is not cloud-agnostic and can be used to deploy resources across a single public cloud

Explanation

Terraform is a cloud-agnostic tool, and therefore isn't limited to a single cloud provider, such as AWS CloudFormation or Azure Resource Manager. Terraform supports all of the major cloud providers and allows IT organizations to focus on learning a single tool for deploying its infrastructure, regardless of what platform it's being deployed on

Question 16: **Correct**
Which flag would be used within a Terraform configuration block to identify the specific version of a provider required?

- required_providers
(Correct)

- required-version
- required-provider
- required_versions

Explanation

For production use, you should constrain the acceptable provider versions via configuration file to ensure that new versions with breaking changes will not be automatically installed by `terraform init` in the future. When `terraform init` is run *without* provider version constraints, it prints a suggested version constraint string for each provider

For example:

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

Question 17: **Correct**
From the answers below, select the advantages of using Infrastructure as Code. (select four)

- Provide a codified workflow to develop customer-facing applications
- Easily integrate with application workflows (GitLab Actions, Azure DevOps, CI/CD tools)
(Correct)
- Safely test modifications using a "dry run" before applying any actual changes
(Correct)
- Provide reusable modules for easy sharing and collaboration
(Correct)
- Easily change and update existing infrastructure
(Correct)

Explanation

Infrastructure as Code is **not** used to develop applications, but it can be used to help deploy or provision those applications to a public cloud provider or on-premises infrastructure.

All of the others are benefits to using Infrastructure as Code over the traditional way of managing infrastructure, regardless if it's public cloud or on-premises.

Question 18: **Correct**
True or False: You can migrate the Terraform backend but only if there are no resources currently being managed.

- True
- False
(Correct)

Explanation

If you are already using Terraform to manage infrastructure, you probably want to transfer to another backend, such as Terraform Cloud, so you can continue managing it. By migrating your Terraform state, you can hand off infrastructure without de-provisioning anything.

Question 19: **Correct**
Using multi-cloud and provider-agnostic tools provides which of the following benefits? (select two)

- slower provisioning speed allows the operations team to catch mistakes before they are applied

- increased risk due to all infrastructure relying on a single tool for management
- can be used across major cloud providers and VM hypervisors
(Correct)
- operations teams only need to learn and manage a single tool to manage infrastructure, regardless of where the infrastructure is deployed
(Correct)

Explanation

Using a tool like Terraform can be advantageous for organizations deploying workloads across multiple public and private cloud environments. Operations teams only need to learn a single tool, single language, and can use the same tooling to enable a DevOps-like experience and workflows.

Question 20: **Correct**
Which of the following represents a feature of Terraform Cloud that is NOT free to customers?

- private module registry
- VCS integration
- workspace management
- roles and team management
(Correct)

Explanation

Information on the comparisons of Terraform Cloud can be found at this link.

Question 21: **Correct**
What are the benefits of using Infrastructure as Code? (select five)

- Infrastructure as Code is relatively simple to learn and write, regardless of a user's prior experience with developing code
(Correct)
- Infrastructure as Code gives the user the ability to recreate an application's infrastructure for disaster recovery scenarios
(Correct)
- Infrastructure as Code easily replaces development languages such as Go and .Net for application development
- Infrastructure as Code allows a user to turn a manual task into a simple, automated deployment
(Correct)
- Infrastructure as Code is easily repeatable, allowing the user to reuse code to deploy similar, yet different resources
(Correct)
- Infrastructure as Code provides configuration consistency and standardization among deployments
(Correct)

Explanation

If you are new to infrastructure as code as a concept, it is the process of managing infrastructure in a file or files rather than manually configuring resources in a user interface. A resource in this instance is any piece of infrastructure in a given environment, such as a virtual machine, security group, network interface, etc.

At a high level, Terraform allows operators to use HCL to author files containing definitions of their desired resources on almost any provider (AWS, GCP,

GitHub, Docker, etc) and automates the creation of those resources at the time of application.

Question 22: **Correct**
True or False: State is a requirement for Terraform to function.

- False
- True
(Correct)

Explanation

Terraform requires some sort of database to map Terraform config to the real world. When you have a resource in your configuration, Terraform uses this map to know how that resource is represented. Therefore, to map configuration to resources in the real world, Terraform uses its own state structure.

Question 23: **Correct**
You have been given requirements to create a security group for a new application. Since your organization standardizes on Terraform, you want to add this new security group with the fewest number of lines of code. What feature could you use to iterate over a list of required tcp ports to add to the new security group?

- dynamic block
(Correct)
- terraform import
- splat expression
- dynamic backend

Explanation

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.

You can find more information on dynamic blocks using this link.

Question 24: **Correct**
True or False: Workspaces provide identical functionality in the open-source, Terraform Cloud, and Enterprise versions of Terraform.

- True
- False
(Correct)

Explanation

Workspaces, managed with the `terraform workspace` command, isn't the same thing as Terraform Cloud's workspaces. Terraform Cloud workspaces act more like completely separate working directories.

CLI workspaces (OSS) are just alternate state files.

Question 25: **Correct**
Select the feature below that best completes the sentence:

The following list represents the different types of _____ available in Terraform.

1. `max`
2. `min`
3. `join`
4. `replace`
5. `list`
6. `length`
7. `range`

- backends

- data sources
- named values
- functions
  (Correct)

  Explanation

  The Terraform language includes a number of built-in functions that you can call from within expressions to transform and combine values. The Terraform language does not support user-defined functions, and only the functions built into the language are available for use.

Question 26: **Correct**
Which of the following best describes a Terraform provider?

- describes an infrastructure object, such as a virtual network, compute instance, or other components
- serves as a parameter for a Terraform module that allows a module to be customized
- a plugin that Terraform uses to translate the API interactions with the service or provider
  (Correct)
- a container for multiple resources that are used together

  Explanation

  A provider is responsible for understanding API interactions and exposing resources. Providers generally are an IaaS (e.g., Alibaba Cloud, AWS, GCP, Microsoft Azure, OpenStack), PaaS (e.g., Heroku), or SaaS services (e.g., Terraform Cloud, DNSimple, CloudFlare).

Question 27: **Correct**
What Terraform feature is shown in the example below?

```
resource "aws_security_group" "example" {
  name = "sg-app-web-01"

  dynamic "ingress" {
    for_each = var.service_ports
    content {
      from_port = ingress.value
      to_port   = ingress.value
      protocol  = "tcp"
    }
  }
}
```

- conditional expression
- local values
- data source
- dynamic block
  (Correct)

  Explanation

  You can dynamically construct repeatable nested blocks like `ingress` using a special `dynamic` block type, which is supported inside `resource`, `data`, `provider`,

and `provisioner` blocks

Question 28: **Correct**
True or False: Multiple providers can be declared within a single Terraform configuration file.

- true
(Correct)

- false

Explanation

Multiple provider blocks can exist if a Terraform configuration is composed of multiple providers, which is a common situation. To add multiple providers in your configuration, declare the providers, and create resources associated with those providers.

Question 29: **Correct**
What Terraform command can be used to inspect the current state file?

Example:

Larger image

- `terraform inspect`

- `terraform read`

- `terraform state`

- `terraform show`
(Correct)

Explanation

The `terraform show` command is used to provide human-readable output from a state or plan file. This can be used to inspect a plan to ensure that the planned operations are expected, or to inspect the current state as Terraform sees it.

Machine-readable output can be generated by adding the `-json` command-line flag.

Note: When using the `-json` command-line flag, any sensitive values in Terraform state will be displayed in plain text.

Question 30: **Correct**
Your organization has moved to AWS and has manually deployed infrastructure using the console. Recently, a decision has been made to standardize on Terraform for all deployments moving forward.

What can you do to ensure that all existing is managed by Terraform moving forward without interruption to existing services?

- using `terraform import`, import the existing infrastructure into your Terraform state
(Correct)

- resources that are manually deployed in the AWS console cannot be imported by Terraform

- delete the existing resources and recreate them using new a Terraform configuration so Terraform can manage them moving forward

- submit a ticket to AWS and ask them to export the state of all existing resources and use `terraform import` to import them into the state file

Explanation

Terraform is able to import existing infrastructure. This allows you to 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 sure you're confident that you can use Terraform in the future if it currently doesn't

support every AWS service or feature you need today.

Question 31: **Correct**
Terraform-specific settings and behaviors are declared in which configuration block type?

- resource
- provider
- data
- terraform
(Correct)

Explanation

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.

Question 32: **Correct**
Which of the following allows Terraform users to apply policy as code to enforce standardized configurations for resources being deployed via infrastructure as code?

- sentinel
(Correct)
- module registry
- functions
- workspaces

Explanation

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.

Question 33: **Correct**
After executing a `terraform apply`, you notice that a resource has a tilde (~) next to it. What does this infer?

- the resource will be destroyed and recreated
- Terraform can't determine how to proceed due to a problem with the state file
- the resource will be created
- the resource will be updated in place
(Correct)

Explanation

The prefix `-/+` means that Terraform will destroy and recreate the resource, rather than updating it in-place. Some attributes and resources can be updated in-place and are shown with the `~` prefix.

Question 34: **Incorrect**
What is the purpose of using the `local-exec` provisioner? (select two)

- executes a command on the resource to invoke an update to the Terraform state
- ensures that the resource is only executed in the local infrastructure where

Terraform is deployed
(Incorrect)

- to invoke a local executable
(Correct)

- to execute one or more commands on the machine running Terraform
(Correct)

Explanation

The `local-exec` provisioner invokes a local executable after a resource is created. This invokes a process on the machine running Terraform, not on the resource.

Note that even though the resource will be fully created when the provisioner is run, there is no guarantee that it will be in an operable state - for example, system services such as `sshd` may not be started yet on compute resources.

Question 35: **Correct**
In regards to Terraform state file, select all the statements below which are correct: (select four)

- storing state remotely can provide better security
(Correct)

- using the `mask` feature, you can instruct Terraform to mask sensitive data in the state file

- the state file is always encrypted at rest

- Terraform Cloud always encrypts state at rest
(Correct)

- when using local state, the state file is stored in plain-text
(Correct)

- the Terraform state can contain sensitive data, therefore the state file should be protected from unauthorized access
(Correct)

Explanation

Terraform state can contain sensitive data, depending on the resources in use and your definition of "sensitive." The state contains resource IDs and all resource attributes. For resources such as databases, this may contain initial passwords.

When using local state, state is stored in plain-text JSON files.

If you manage any sensitive data with Terraform (like database passwords, user passwords, or private keys), treat the state itself as sensitive data.

Storing Terraform state remotely can provide better security. As of Terraform 0.9, Terraform does not persist state to the local disk when remote state is in use, and some backends can be configured to encrypt the state data at rest.

Question 36: **Correct**
What are some of the features of Terraform state? (select three)

- inspection of cloud resources

- increased performance
(Correct)

- determining the correct order to destroy resources
(Correct)

- mapping configuration to real-world resources
(Correct)

Explanation

See this page on the purpose of Terraform state and the benefits it provides.

Question 37: **Correct**
What is the best and easiest way for Terraform to read and write secrets from HashiCorp Vault?

- integration with a tool like Jenkins
- API access using the AppRole auth method
- Vault provider
(Correct)
- CLI access from the same machine running Terraform

Explanation

The Vault provider allows Terraform to read from, write to, and configure Hashicorp Vault.

Question 38: **Correct**
In the example below, where is the value of the DNS record's IP address originating from?

```
resource "aws_route53_record" "www" {
  zone_id = aws_route53_zone.primary.zone_id
  name    = "www.helloworld.com"
  type    = "A"
  ttl     = "300"
  records = [module.web_server.instance_ip_addr]
}
```

- the regular expression named module.web_server
- the output of a module named web_server
(Correct)
- value of the web_server parameter from the variables.tf file
- by querying the AWS EC2 API to retrieve the IP address

Explanation

In a parent module, outputs of child modules are available in expressions as `module.<MODULE NAME>.<OUTPUT NAME>`. For example, if a child module named `web_server` declared an output named `instance_ip_addr`, you could access that value as `module.web_server.instance_ip_addr`.

Question 39: **Correct**
True or False: Provisioners should only be used as a last resort.

- true
(Correct)
- false

Explanation

Provisioners are used to execute scripts on a local or remote machine as part of resource creation or destruction. Provisioners can be used to bootstrap a resource, cleanup before destroy, run configuration management, etc. Even if the functionality you need is not available in a provider today, HashiCorp suggests that you consider `local-exec` usage as a temporary workaround and to open an issue in the relevant provider's repo to discuss adding first-class support.

Question 40: **Correct**
Which Terraform command will force a marked resource to be destroyed and recreated on the next apply?

-

`terraform refresh`

- `terraform destroy`
- `terraform fmt`
- `terraform taint`
(Correct)

Explanation

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. The next `terraform apply` will implement this change.

Question 41: **Correct**
In Terraform Enterprise, a workspace can be mapped to how many VCS (version control systems) repos?

- 3
- 1
(Correct)
- 2
- 5

Explanation

A workspace can only be configured to a single VCS repo, however, multiple workspaces can use the same repo, if needed. A good explanation of how to configure your code repositories can be found here.

Question 42: **Correct**
In order to make a Terraform configuration file dynamic and/or reusable, static values should be converted to use what?

- regular expressions
- output value
- module
- input parameters
(Correct)

Explanation

Input variables serve as parameters for a Terraform module, allowing aspects of the module to be customized without altering the module's own source code, and allowing modules to be shared between different configurations.

Question 43: **Incorrect**
Which of the following Terraform files should be ignored by Git when committing code to a repo? (select two)

- output.tf
- terraform.tfvars
(Correct)
- variables.tf
(Incorrect)
-

terraform.tfstate
(Correct)

Explanation

The .gitignore file should be configured to ignore Terraform files that either contain sensitive data or aren't required to save.

The terraform.tfstate file contains the terraform state of a specific environment and doesn't need to be preserved in a repo. The terraform.tfvars file may contain sensitive data, such as passwords or IP addresses of an environment that you may not want to share with others.

Question 44: **Correct**
Given the Terraform configuration below, in which order will the resources be created?

```
resource "aws_instance" "web_server" {
    ami = "i-abdce12345"
    instance_type = "t2.micro"
}

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

- no resources will be created
- aws_instance will be created first
aws_eip will be created second
(Correct)
- aws_eip will be created first
aws_instance will be created second
- resources will be created simultaneously

Explanation

The aws_instance will be created first, and then aws_eip will be created second due to the aws_eip's resource dependency of the aws_instance id

More information on resource dependencies can be found at this link.

Question 45: **Correct**
When configuring a remote backend in Terraform, it might be a good idea to purposely omit some of the required arguments to ensure secrets and other relevant data are not inadvertently shared with others. What are the ways the remaining configuration can be added to Terraform so it can initialize and communicate with the backend? (select three)

- command-line key/value pairs
(Correct)
- directly querying HashiCorp Vault for the secrets
- use the -backend-config=PATH to specify a separate config file
(Correct)
- interactively on the command line
(Correct)

Explanation

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*.

With a partial configuration, the remaining configuration arguments must be provided as part of the initialization process. There are several ways to supply the remaining arguments:

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`.

Question 46: **Correct**
What happens when a `terraform apply` command is executed?

- creates the execution plan for the deployment of resources
- applies the changes required in the target infrastructure in order to reach the desired configuration
(Correct)
- reconciles the state Terraform knows about with the real-world infrastructure
- the backend is initialized and the working directory is prepped

Explanation

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.

Question 47: **Correct**
After running into issues with Terraform, you need to enable verbose logging to assist with troubleshooting the error. Which of the following values provides the MOST verbose logging?

- DEBUG
- WARN
- TRACE
(Correct)
- INFO
- ERROR

Explanation

Terraform has detailed logs that 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.

Question 48: **Correct**
What does the command `terraform fmt` do?

- deletes the existing configuration file
- rewrite Terraform configuration files to a canonical format and style
(Correct)
- updates the font of the configuration file to the official font supported by HashiCorp
- formats the state file in order to ensure the latest state of resources can be

obtained

Explanation

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.

Other Terraform commands that generate Terraform configuration will produce configuration files that conform to the style imposed by `terraform fmt`, so using this style in your own files will ensure consistency.

Question 49: **Correct**
You want to use `terraform import` to start managing infrastructure that was not originally provisioned through infrastructure as code. Before you can import the resource's current state, what must you do in order to prepare to manage these resources using Terraform?

- update the configuration file to include the new resources **(Correct)**
- modify the Terraform state file to add the new resources
- shut down or stop using the resources being imported so no changes are inadvertently missed
- run `terraform refresh` to ensure that the state file has the latest information for existing resources.

Explanation

The current implementation of Terraform import can only import resources into the state. It does not generate a configuration. Because of this, and prior to running `terraform import`, it is necessary to manually write a `resource` configuration block for the resource to which the imported object will be mapped.

First, add the resources to the configuration file:

```
resource "aws_instance" "example" {
  # ...instance configuration...
}
```

Then run the following command:

```
$ terraform import aws_instance.example i-abcd1234
```

Question 50: **Incorrect**
What is a downside to using a Terraform provider, such as the Vault provider, to interact with sensitive data, such as reading secrets from Vault?

- secrets are persisted to the state file and plans **(Correct)**
- Terraform requires a unique auth method to work with Vault **(Incorrect)**
- Terraform and Vault must be running on the same physical host
- Terraform and Vault must be running on the same version

Explanation

Interacting with Vault from Terraform causes any secrets that you read and write to be persisted in both Terraform's state file *and* in any generated plan files. For any Terraform module that reads or writes Vault secrets, these files should be treated as sensitive and protected accordingly.

Retake test
Continue
Fullscreen

HashiCorp Terraform - Practice Exam #2 - Results

Return to review

Attempt 1

All questions

Question 1: **Incorrect**

Which of the following variable declarations is going to result in an error?

-

```
variable "examp
variable "example" {
  description = "This is a variable description"
  type        = list(string)
  default     = {}
}
```

(Correct)

-

```
variable "example" {
  description = "This is a test"
  type        = map
  default     = {"one" = 1, "two" = 2, "Three" = "3"}
}
```

-

```
variable "example" {
  type = object({})
}
```

(Incorrect)

Explanation

Lists are defined with [ ], maps are defined with { }.

https://www.terraform.io/docs/configuration/types.html#structural-types

Question 2: **Correct**

In terraform, most resource dependencies are handled automatically. Which of the following statements describes best how terraform resource dependencies are handled?

-

Resource dependencies are identified and maintained in a file called `resource.dependencies`. Each terraform provider is required to maintain a list of all resource dependencies for the provider and it's included with the plugin during initialization when `terraform init` is executed. The file is located in the `terraform.d` folder.

-

Terraform analyses any expressions within a resource block to find references to other objects and treats those references as implicit ordering requirements when creating, updating, or destroying resources.
(Correct)

-

Resource dependencies are handled automatically by the `depends_on` meta_argument, which is set to true by default.

-

The terraform binary contains a built-in reference map of all defined Terraform resource dependencies. Updates to this dependency map are reflected in terraform versions. To ensure you are working with the latest resource dependency map you much be running the latest version of Terraform.

Explanation

Terraform analyses any expressions within a resource block to find references to other objects and treats those references as implicit ordering requirements when creating, updating, or destroying resources.

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

Question 3: **Correct**

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"
  }
}
```

- 1.3.1
- 1.2.9
  (Correct)
- 1.3.0
- 1.2.3
  (Correct)

Explanation

`~> 1.2.0` will match any non-beta version of the provider between `>= 1.2.0` and `< 1.3.0`. For example, `1.2.X`

https://www.terraform.io/docs/configuration/modules.html#gt-1-2-0-1

Question 4: **Correct**

Choose the correct answer which fixes the syntax of the following Terraform code:

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

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

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

Explanation

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

Question 5: **Correct**
Terraform Enterprise (also referred to as pTFE) requires what type of backend database for a clustered deployment?

- MSSQL
- PostgreSQL
(Correct)
- Cassandra
- MySQL

Explanation

External Services mode stores the majority of the stateful data used by the instance in an external PostgreSQL database and an external S3-compatible endpoint or Azure blob storage. There is still critical data stored on the instance that must be managed with snapshots. Be sure to check the PostgreSQL Requirements for information that needs to be present for Terraform Enterprise to work. This option is best for users with expertise managing PostgreSQL or users that have access to managed PostgreSQL offerings like AWS RDS.

Question 6: **Correct**
Select all features which are exclusive to Terraform Enterprise. (select three)

- Audit Logs
(Correct)
- Cost Estimation
- Clustering
(Correct)
- Sentinel
- SAML/SSO
(Correct)

Explanation

Sentinel and Cost Estimation are both available in Terraform Cloud, though not at the free tier level.

https://www.hashicorp.com/products/terraform/pricing/

Question 7: **Correct**
A user has created three workspaces using the command line - prod, dev, and test. The user wants to create a fourth workspace named `stage`. Which command will the user execute to accomplish this?

- terraform workspace -new stage
- terraform workspace create stage
- terraform workspace new stage
(Correct)
- terraform workspace -create stage

Explanation

The `terraform workspace new` command is used to create a new workspace.
https://www.terraform.io/docs/commands/workspace/new.html

Question 8: **Correct**
Which statements best describes what the local variable assignment is doing in the following code snippet:

```
variable "subnet_details" {
  type = list(object({
    cidr             = string
    subnet_name      = string
    route_table_name = string
    aznum            = number
  }))
}

locals {
  route_tables_all = distinct([for s in var.subnet_details : s.route_table_name ])
}
```

- Create a map of route table names from a list of subnet names
- Create a distinct list of route table name objects
- Create a map of route table names to subnet names
- Create a list of route table names eliminating duplicates
  (Correct)

  Explanation

  `route_tables_all` is assigned a list of unique route table names filtered from a list of objects describing subnet details, one of those object attributes being `route_table_name`.

  Question 9: **Correct**
  What feature of Terraform Cloud and/or Terraform Enterprise can you publish and maintain a set of custom modules which can be used within your organization?

- private module registry
  (Correct)
- remote runs
- custom VCS integration
- 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.

  Question 10:

  When multiple arguments with single-line values appear on consecutive lines at the same nesting level, HashiCorp recommends that you:

- put arguments in alphabetical order
  1. `name = "www.example.com"`
  2. `records = [aws_eip.lb.public_ip]`
  3. `type = "A"`
  4. `ttl = "300"`
  5. `zone_id = aws_route53_zone.primary.zone_id`
- place all arguments using a variable at the top
  1. `ami = var.aws_ami`
  2. `instance_type = var.instance_size`
  3. `subnet_id = "subnet-0bb1c79de3EXAMPLE"`

4. `tags = {`
5. `  Name = "HelloWorld"`
6. `}`

(Incorrect)

- place a space in between each line
  1. `type = "A"`
  2.
  3. `ttl = "300"`
  4.
  5. `zone_id = aws_route53_zone.primary.zone_id`

- align their equals signs
  1. `ami           = "abc123"`
  2. `instance_type = "t2.micro"`

(Correct)

Explanation

HashiCorp style conventions suggest you that align the equals sign for consecutive arguments for easing readability for configurations

```
ami           = "abc123"
instance_type = "t2.micro"
```

Question 11: **Incorrect**
Complete the following sentence:

For local state, the workspaces are stored directly in a...

- directory called terraform.tfstate.d
  (Correct)

- a file called terraform.tfstate
  (Incorrect)

- a file called terraform.tfstate.backup

- directory called terraform.workspaces.tfstate

Explanation

For local state, Terraform stores the workspace states in a directory called `terraform.tfstate.d`.
https://www.terraform.io/docs/state/workspaces.html#workspace-internals

Question 12: **Correct**
When Terraform needs to be installed in a location where it does not have internet access to download the installer and upgrades, the installation is generally known as to be _____.

- disconnected

- a private install

- air-gapped
  (Correct)

- non-traditional

Explanation

A Terraform Enterprise install that is provisioned on a network that does not have Internet access is generally known as an air-gapped install. These types of installs require you to pull updates, providers, etc. from external sources vs. being able to download them directly.

Question 13: **Correct**
What is the result of the following terraform function call?

```
> zipmap(["a", "b"], [1, 2])
```

-

  {
  "a" = 1
  "b" = 2
  }
  (Correct)

-

  [
  "a" = 1
  "b" = 2
  ]

-

  [
  "a",
  "b",
  "1",
  "2",
  ]

-

  {
  "a",
  "b",
  "1",
  "2",
  }

Explanation

`zipmap` constructs a map from a list of keys and a corresponding list of values. A map is denoted by { } whereas a list is donated by [ ].

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

Question 14: **Correct**
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 `terraform plan` and state files.

-

  True

-

  False
  (Correct)

Explanation

Currently, Terraform has no mechanism to redact or protect secrets that are returned via data sources, so secrets read via this provider will be persisted into the Terraform state, into any plan files, and in some cases in the console output produced while planning and applying. These artifacts must, therefore, all be protected accordingly.

Question 15: **Correct**
Which of the following terraform subcommands could be used to remove the lock on the state for the current configuration?

-

  force-unlock
  (Correct)

-

  state-unlock

-

  Removing the lock on a state file is not possible

-

  unlock

Explanation

`terraform force-unlock` removes the lock on the state for the current configuration. https://www.terraform.io/docs/commands/force-unlock.html

Question 16: **Correct**
In the following code snippet, the `block type` is identified by which string?

58. `resource "aws_instance" "db" {`

```
  ami           = "ami-123456"
  instance_type = "t2.micro"
}
```

- "db"
- "aws_instance"
- instance_type
- resource
  (Correct)

  Explanation

  The format of resource block configurations is as follows:

  <block type> "<resource type>" "<local name/label>"

  Question 17: **Correct**
  A user runs `terraform init` on their RHEL based server and per the output, two provider plugins are downloaded:

```
$ terraform init

Initializing the backend...

Initializing provider plugins...
- Checking for available provider plugins...
- Downloading plugin for provider "aws" (hashicorp/aws) 2.44.0...
- Downloading plugin for provider "random" (hashicorp/random) 2.2.1...

Terraform has been successfully initialized!
```

  Where are these plugins downloaded to?

- The .terraform/plugins directory in the directory terraform init was executed in.
  (Correct)
- The .terraform.plugins directory in the directory terraform init was executed in.
- The .terraform.d directory in the directory terraform init was executed in.
- /etc/terraform/plugins

  Explanation

  By default, `terraform init` downloads plugins into a subdirectory of the working directory, .terraform/plugins, so that each working directory is self-contained.

  Question 18: **Correct**
  Which of the following is not a valid Terraform string function?

- tostring
  (Correct)
- format
- replace
- join

  Explanation

  tostring is not a string function, it is a type conversion function. `tostring` converts its argument to a string value.

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

Question 19: **Correct**
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]
}
```

- The AMI used for the EC2 instance
- The S3 bucket labeled company_data
- The EIP with an id of ami-2757f631
- The EC2 instance labeled web_server
(Correct)

Explanation

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.example` is an explicit dependency.

Question 20: **Correct**
Environment variables can be used to set variables. The environment variables must be in the format `"____"_<variablename>`. Select the correct prefix string from the following list.

- TF_VAR
(Correct)
- TF_VAR_NAME
- TF_ENV
- TF_ENV_VAR

Explanation

Environment variables can be used to set variables. The environment variables must be in the format `TF_VAR_name` and this will be checked last for a value. For example:

```
export TF_VAR_region=us-west-1
export TF_VAR_ami=ami-049d8641
export TF_VAR_alist='[1,2,3]'
export TF_VAR_amap='{ foo = "bar", baz = "qux" }'
```

https://www.terraform.io/docs/commands/environment-variables.html

Question 21: **Correct**
While Terraform is generally written using the HashiCorp Configuration Language (HCL), what other syntax can Terraform be expressed in?

- TypeScript
- YAML
-

XML

- JSON
(Correct)

Explanation

The constructs in the Terraform language can also be expressed in JSON syntax, which is harder for humans to read and edit but easier to generate and parse programmatically.

Question 22: **Correct**
What is the result of the following terraform function call?

```
> index(["a", "b", "c"], "c")
```

- true
- 0
- 1
- 2
(Correct)

Explanation

`index` finds the element index for a given value in a list starting with index 0.

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

Question 23: **Incorrect**
When using providers that require the retrieval of data, such as the HashiCorp Vault provider, in what phase does Terraform actually retrieve the data required?

- `terraform init`
(Incorrect)
- `terraform delete`
- `terraform apply`
- `terraform plan`
(Correct)

Explanation

It is important to consider that Terraform reads from data sources during the `plan` phase and writes the result into the plan. For something like a Vault token which has an explicit TTL, the `apply` must be run before the data, or token, in this case, expires, otherwise, Terraform will fail during the apply phase.

Question 24: **Correct**
Which Terraform command will check and report errors within modules, attribute names, and value types to make sure they are syntactically valid and internally consistent?

- `terraform fmt`
- `terraform validate`
(Correct)
- `terraform show`
- `terraform format`

Explanation

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 the correctness of attribute names and value types.

Question 25: **Correct**
True or False? `terraform init` cannot automatically download community providers.

Larger image

- False
(Correct)

- True

Explanation

With Terraform 0.13, `terraform init` can now automatically download community providers.

https://www.hashicorp.com/blog/automatic-installation-of-third-party-providers-with-terraform-0-13

Question 26: **Correct**
When writing Terraform code, HashiCorp recommends that you use how many spaces between each nesting level?

- 2
(Correct)

- 5

- 1

- 4

Explanation

HashiCorp style conventions state that you should use 2 spaces between each nesting level to improve the readability of Terraform configurations.

Check this link for more information

Question 27: **Correct**
The following is a snippet from a Terraform configuration file:

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

provider "aws" {
  region = "us-west-1"
}
```

which, when validated, results in the following error:-

```
Error: Duplicate provider configuration

  on main.tf line 5:
   5: provider "aws" {

A default provider configuration for "aws" was already given at
main.tf:1,1-15. If multiple configurations are required, set the "______"
argument for alternative configurations.
```

Fill in the blank in the error message with the correct string from the list below.

- alias
(Correct)

- label
- multi
- version

Explanation

An `alias` meta-arguement is used when using the same provider with different configurations for different resources.

https://www.terraform.io/docs/configuration/providers.html#alias-multiple-provider-instances

Question 28: **Correct**
What is the result of the following terraform function call?

```
> lookup({a="hello", b="goodbye"}, "c", "what?")
```

- hello
- goodbye
- what?
(Correct)
- c

Explanation

`lookup` retrieves the value of a single element from a map, given its key. If the given key does not exist, a the given default value is returned instead. In this case, the function call is searching for the key "c". Because there there is no key "c", the default vault "what?" is returned.

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

Question 29: **Correct**
True or False? By default, Terraform `destroy` will prompt for confirmation before proceeding.

- False
- True
(Correct)

Explanation

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:
```

Question 30: **Correct**
The Terraform language supports a number of different syntaxes for comments. Select all that are supported. (select three)

- #
(Correct)
- <* and *>
- //
(Correct)

- /* and */
  (Correct)

Explanation

Terraform supports the #, //, and /*..*/ for commenting Terraform configuration files. Please use them when writing Terraform so both you and others who are using your code have a full understanding of what the code is intended to do.

https://www.terraform.io/docs/configuration/syntax.html#comments

Question 31: **Correct**
In order to reduce the time it takes to provision resources, Terraform uses parallelism. By default, how many resources will Terraform provision concurrently?

- 50
- 10
  (Correct)
- 20
- 5

Explanation

Terraform can limit the number of concurrent operations as Terraform walks the graph using the `-parallelism=n` argument. The default value for this setting is `10`. This setting might be helpful if you're running into API rate limits.

Question 32: **Correct**
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?

- `terraform workspace select dev`
  (Correct)
- `terraform workspace dev`
- `terraform workspace switch dev`
- `terraform workspace -switch dev`

Explanation

The `terraform workspace select` command is used to choose a different workspace to use for further operations.
https://www.terraform.io/docs/commands/workspace/select.html

Question 33: **Correct**
A user has created a module called "*my_test_module*" and committed it to GitHub. Over time, several commits have been made with updates to the module, each tagged in GitHub with an incremental version number. Which of the following lines would be required in a module configuration block in terraform to select tagged version v1.0.4?

- source = "git::https://example.com/my_test_module.git@tag=v1.0.4"
- source = "git::https://example.com/my_test_module.git#tag=v1.0.4"
- source = "git::https://example.com/my_test_module.git&ref=v1.0.4"
- source = "git::https://example.com/my_test_module.git?ref=v1.0.4"
  (Correct)

Explanation

By default, Terraform will clone and use the default branch (referenced by `HEAD`) in the selected repository. You can override this using the `ref` argument:

```
module "vpc" {
  source = "git::https://example.com/vpc.git?ref=v1.2.0"
}
```

The value of the `ref` argument can be any reference that would be accepted by the `git checkout` command, including branch and tag names.

https://www.terraform.io/docs/modules/sources.html#selecting-a-revision

Question 34: **Correct**
Select the most accurate statement to describe the Terraform language from the following list.

- Terraform is an immutable, procedural, Infrastructure as Code configuration management language based on Hashicorp Configuration Language, or optionally JSON.
- Terraform is a mutable, declarative, Infrastructure as Code configuration management language based on Hashicorp Configuration Language, or optionally JSON.
- Terraform is a mutable, procedural, Infrastructure as Code provisioning language based on Hashicorp Configuration Language, or optionally YAML.
- Terraform is an immutable, declarative, Infrastructure as Code provisioning language based on Hashicorp Configuration Language, or optionally JSON.
(Correct)

Explanation

Terraform is not a configuration management tool - https://www.terraform.io/intro/vs/chef-puppet.html

Terraform is a declarative language - https://www.terraform.io/docs/configuration/index.html

Terraform supports a syntax that is JSON compatible - https://www.terraform.io/docs/configuration/syntax-json.html

Terraform is primarily designed on immutable infrastructure principles - https://www.hashicorp.com/resources/what-is-mutable-vs-immutable-infrastructure

Question 35: **Incorrect**
Which of the following statements best describes the Terraform `list(...)` type?

- a sequence of values identified by consecutive whole numbers starting with zero.
(Correct)
- a collection of values where each is identified by a string label.
(Incorrect)
- a collection of unique values that do not have any secondary identifiers or ordering.
- a collection of named attributes that each have their own type.

Explanation

A terraform list is a sequence of values identified by consecutive whole numbers starting with zero.

https://www.terraform.io/docs/configuration/types.html#structural-types

Question 36: **Incorrect**
Select the operating systems which are supported for a clustered Terraform Enterprise: (select four)

- Ubuntu
(Correct)
-

Amazon Linux
(Correct)

- CentOS
(Correct)

- Unix
(Incorrect)

- Red Hat
(Correct)

Explanation

Note: (5/27/20) This question has been recently updated to reflect documentation updates on the HashiCorp website. It seems they have removed the clustering-specific requirements and are now following the standard Enterprise operating system requirements.

Terraform Enterprise currently supports running under the following operating systems for a Clustered deployment:

- Ubuntu 16.04.3 - 16.04.5 / 18.04

- Red Hat Enterprise Linux 7.4 through 7.7

- CentOS 7.4 - 7.7

- Amazon Linux

- Oracle Linux

Clusters currently don't support other Linux variants.

https://www.terraform.io/docs/enterprise/before-installing/index.html#operating-system-requirements

Question 37: **Incorrect**
Which of the following commands will launch the Interactive console for Terraform interpolations?

- terraform console
(Correct)

- terraform

- terraform cli
(Incorrect)

- terraform cmdline

Explanation

The `terraform console` command provides an interactive console for evaluating expressions.

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

Question 38: **Correct**
Complete the following sentence:

The terraform `state` command can be used to ____

- refresh the existing state

- view the entire state file

- modify the current state, such as removing items
(Correct)

- There is no such command

Explanation

The `terraform state` command is used for advanced state management. 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

Question 39: **Correct**
Terraform has detailed logs which can be enabled by setting the _____ environmental variable.

- TF_TRACE
- TF_LOG
(Correct)
- TF_DEBUG
- TF_INFO

Explanation

Terraform has detailed logs that 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.

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

Question 40: **Correct**
Anyone can publish and share modules on the `Terraform Public Module Registry`, and meeting the requirements for publishing a module is extremely easy. Select from the following list all valid requirements. (select three)

- The module must be PCI/HIPPA compliant.
- The registry uses tags to identify module versions. Release tag names must be for the format *x.y.z*, and can optionally be prefixed with a `v` .
(Correct)
- The module must be on GitHub and must be a public repo.
(Correct)
- Module repositories must use this three-part name format, *terraform-<PROVIDER>-<NAME>*.
(Correct)

Explanation

The list below contains all the requirements for publishing a module. Meeting the requirements for publishing a module is extremely easy. The list may appear long only to ensure we're detailed, but adhering to the requirements should happen naturally.

GitHub. The module must be on GitHub and must be a public repo. This is only a requirement for the public registry. If you're using a private registry, you may ignore this requirement.

Named `terraform-<PROVIDER>-<NAME>`. Module repositories must use this three-part name format, where `<NAME>` reflects the type of infrastructure the module manages and `<PROVIDER>` is the main provider where it creates that infrastructure. The `<NAME>` segment can contain additional hyphens. Examples: `terraform-google-vault` or `terraform-aws-ec2-instance`.

Repository description. The GitHub repository description is used to populate the short description of the module. This should be a simple one-sentence description of the module.

Standard module structure. The module must adhere to the standard module structure. This allows the registry to inspect your module and generate documentation, track resource usage, parse submodules and examples, and more.

`x.y.z` tags for releases. The registry uses tags to identify module versions. Release tag names must be a semantic version, which can optionally be prefixed with a `v`. For example, `v1.0.4` and `0.9.2`. To publish a module initially, at least one release tag must be present. Tags that don't look like version numbers are

ignored.

https://www.terraform.io/docs/registry/modules/publish.html#requirements

Question 41: **Incorrect**
Select all Operating Systems that Terraform is available for. (select five)

- macOS
(Correct)
- Unix
(Incorrect)
- Solaris
(Correct)
- FreeBSD
(Correct)
- Windows
(Correct)
- Linux
(Correct)

Explanation

Terraform is available for macOS, FreeBSD, OpenBSD, Linux, Solaris, Windows

https://www.terraform.io/downloads.html

Question 42: **Correct**
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?

- artifactory
- terraform enterprise
- github
(Correct)
- s3
- consul

Explanation

github is not a supported backend type.
https://www.terraform.io/docs/backends/types/index.html

Question 43: **Correct**
True or False. The `terraform refresh` command is used to reconcile the state Terraform knows about (via its state file) with the real-world infrastructure. If drift is detected between the real-world infrastructure and the last known-state, it will modify the infrastructure to correct the drift.

- True
- False
(Correct)

Explanation

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.

This 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.

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

Question 44: **Correct**
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]
}
```

- non-dependency resource
- explicit dependency
  (Correct)
- implicit dependency
- internal dependency

Explanation

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.

Question 45: **Incorrect**
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)

- GitHub
  (Correct)
- CVS Version Control
  (Incorrect)
- GitHub Enterprise
  (Correct)
- Bitbucket Cloud
  (Correct)
- Azure DevOps Server
  (Correct)

Explanation

Terraform Cloud supports the following VCS providers:

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

https://www.terraform.io/docs/cloud/vcs/index.html#supported-vcs-providers

Question 46: **Incorrect**
Why might a user opt to include the following snippet in their configuration file?

```
terraform {
```

```
  required_version = ">= 0.12"
}
```

- The user wants to ensure that the application being deployed is a minimum version of 0.12
- versions before Terraform 0.12 were not approved by HashiCorp to be used in production
- Terraform 0.12 introduced substantial changes to the syntax used to write Terraform configuration
(Correct)
- this ensures that all Terraform providers are above a certain version to match the application being deployed
(Incorrect)

Explanation

You can use `required_version` to ensure that a user deploying infrastructure is using Terraform 0.12 or greater, due to the vast number of changes that were introduced. As a result, many previously written configurations had to be converted or rewritten.

Question 47: **Incorrect**----------

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

- Explicit use of a provider block in configuration, optionally including a version constraint.
(Correct)
- Use of any resource belonging to a particular provider in a resource or data block in the configuration.
(Correct)
- Existence of any provider plugins found locally in the working directory.
(Incorrect)
- Existence of any resource instance belonging to a particular provider in the current *state*.
(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://www.terraform.io/docs/commands/providers.html

Question 48: **Correct**

Which of the following is an *invalid* variable name?

- instance_name
- web
- count
(Correct)
- var1

Explanation

`count` is a reserved word. The `count` parameter on resources can simplify configurations and let you scale resources by simply incrementing a number.

https://www.terraform.io/intro/examples/count.html

Question 49: **Correct**
True or False? Each Terraform workspace uses its own state file to manage the infrastructure associated with that particular workspace.

- False
- True
(Correct)

Explanation

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.

Question 50: **Incorrect**
During a `terraform apply`, a resource is successfully created but eventually fails during provisioning. What happens to the resource?

- the resource is marked as tainted
(Correct)
- it is automatically deleted
(Incorrect)
- the terraform plan is rolled back and all provisioned resources are removed
- Terraform attempts to provision the resource up to three times before exiting with an error

Explanation

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.

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.

Retake testContinue
Fullscreen

Terraform Associate Certification - Results

Return to review

Attempt 1

All knowledge areas

All questions

Question 1: **Correct**
What provisioner can you use to that invokes a command on the machine that runs Terraform CLI?

- local-cmd
- remote-exec
- exec-local
- local-exec
(Correct)
- salt-masterless

Explanation

local-exec Provisioner

The `local-exec` provisioner invokes a local executable after a resource is created. This invokes a process on the machine running Terraform, not on the resource.

Example:

```
resource "aws_instance" "quiz_experts" {
  # ...

  provisioner "local-exec" {
    command = "echo ${aws_instance.quiz_experts.private_ip} >> private_ips.txt"
  }
}
```

Note: Provisioners should only be used as a last resort.

Click here to know more about local-exec provisioner

Question 2: **Correct**

Why does the traditional approach of managing infrastructure not go well with large-scale, distributed systems architectures? Choose *THREE* correct answers.

- The traditional approach can't keep up with higher elasticity of infrastructure, where resources might live days to weeks instead of months to years. **(Correct)**
- The traditional approach was okay if we didn't have to manage a lot of infrastructure. But it does not fit well where The scale of infrastructure is much larger, because instead of a handful of large instances, we might have many smaller instances, so there are many more things we need to provision. **(Correct)**
- Infrastructure changes are now relatively minimal, whereas the traditional approach helps managing infrastructures where changes are more often.
- Logging it into a management portal or an administrative console, and pointing and clicking to provision large infrastructure is slow and prone to human error. **(Correct)**
- Pointing and clicking into an administrative console is a more scalable solution for large-scale, distributed systems architectures

Explanation

Let us take an example to understand it. Your organization runs VMware, inside a private data center, and you need a new VMs. The **traditional approach** was: You would file a ticket, and then someone on the other end of this ticketing queue is pulling it off, logging it into a management portal or an administrative console, and pointing and clicking to provision that VM.

This was okay if you didn't have to manage a lot of infrastructure.

Now, as we're making a transition to a **cloud environment**, which is predominantly API-driven, the scale of infrastructure is much larger and more elastic. Instead of a handful of large instances, we might have many smaller instances, so there are many more things we need to provision, and this infrastructure tends to be cyclic. We might scale up to handle a load during a peak day, and then scale down at night to save on cost, because it's not a fixed cost. Unlike owning hardware, we're now paying by the hour. So, it makes sense only to use the infrastructure you need. You have to have this elasticity if you were using the traditional approach to handle the elastic nature of infrastructure. You have to file a thousand tickets every morning to spin up to peak capacity, then file another thousand tickets at night to spin back down, and then manually manage all of this. It slows down the process, and pointing and clicking are prone to human error.

Click here to see a video to know more

Question 3: **Correct**
You can set variables declared in the root module using CLI options and environment variables. But to set variables declared in child modules, you should pass their values in the module block.

- False

- True
(Correct)

Explanation

When you declare variables in the root module of your configuration, you can set their values using CLI options and environment variables. When you declare them in child modules, the calling module should pass values in the `module` block.

Assigning Values to Root Module Variables

When variables are declared in the root module of your configuration, they can be set in a number of ways:

* In a Terraform Cloud workspace.

* Individually, with the `-var` command-line option.

* In variable definitions (`.tfvars`) files, either specified on the command line or automatically loaded.

* As environment variables.

Assigning Values to Child Module Variables

To *call* a module means to include the contents of that module into the configuration with specific values for its input variables. Modules are called from within other modules using `module` blocks:

```
module "servers" {
  source = "./app-cluster"

  # servers and image_id are child module input variables
  servers  = 5
  image_id = "ami-eijo28"
}
```

Click here to know more about assigning values to root module variables

Question 4: **Incorrect**
Let's say a data center went down completely, and we need to failover to a different datacenter or a cloud environment. How does Terraform can help in this scenario? Choose *TWO* correct answers.

- Terraform can be an essential part of your disaster recovery strategy because it helps you stand up new infrastructure very quickly and efficiently.
(Correct)

- Terraform allows cheaper and much more dynamic disaster recovery plans because you can literally build a DR environment on the fly; hence it may not make sense to keep it running all the time and pay a massive bill for it.
(Correct)

- Terraform backs up the data and use various strategies to help you restore databases and everything you need to keep your environments up and running during disaster recovery.

- During disaster recovery, Terraform can help you very quickly stand up a new infrastructure or new production environments in a different region of the same cloud provider in a very short period of time; however, Terraform may charge a small amount for a cross-cloud provider DR.
(Incorrect)

Explanation

Disaster Recovery with Terraform

Let's say the datacenter went completely down and we need to failover to a different datacenter or a cloud environment. You can use Terraform to very quickly stand up all the application infrastructure and get everything ready. So Terraform can be a core piece of your DR recovery plans.

One thing that Terraform won't do, though, is back up your data, so it's very important that you have a good backup strategy to use along with Terraform so that, once your servers and applications are up and running, you also have the databases and everything that you need to keep your environments up and running.

Practice makes perfect, even with DR

Another thing to keep in mind is that a DR plan without practice is not really a DR plan. You should be doing regular tests of your DR strategy just to make sure that, when things really go wrong, you know it will work because you tested it last week or last month. Terraform makes this very easy. You can use Terraform to stand up copies of your environment and do practice runs, like a fire drill. Make sure that your DR plans work before the disaster happens and not after.

Terraform can be an essential part of your disaster recovery strategy because it helps you stand up new infrastructure very quickly and efficiently.

Click here to know more about DR with Terraform

Question 5: **Incorrect**
What are the advantages of using Infrastructure as Code?
Choose *FIVE* correct answers.

- With Infrastructure as Code, users can write configuration once and then reuse it many times.
(Correct)

- Infrastructure as Code can be an essential part of your disaster recovery strategy. It helps you stand up new infrastructure quickly and efficiently.
(Correct)

- Infrastructure as Code provides version-control of infrastructure configuration to provide a single source of truth.
(Correct)

- Infrastructure as Code provides the ability to gracefully roll the infrastructure back to a last known good state.
(Correct)

- Infrastructure as Code is turning the Ops team into developers.
(Incorrect)

- Infrastructure as Code codifies manual pointing and clicking on management console and automate it.
(Correct)

Explanation

Infrastructure as Code

Infrastructure as Code is the process of managing infrastructure in a file or files rather than manually configuring resources in a user interface. A resource in this instance is any piece of infrastructure in a given environment, such as a virtual machine, security group, network interface, etc.

Infrastructure as Code (IaC) is the management of infrastructure (networks, virtual machines, load balancers, and connection topology) in a descriptive model, using the same versioning as DevOps team uses for source code. Like the principle that the same source code generates the same binary, an IaC model generates the same environment every time it is applied. IaC is a key DevOps practice and is used in conjunction with Continuous Delivery.

At a high level, Terraform allows operators to use HCL to author files containing definitions of their desired resources on almost any provider (AWS, GCP, GitHub, Docker, etc) and automates the creation of those resources at the time of apply.

Benefits of infrastructure as code

Infrastructure as code brings a lot of benefits -

Visibility: An infrastructure as code template serves as a very clear reference of what resources are on your account, and what their settings are. You don't have to navigate to the web console to check the parameters.

Stability: If you accidentally change the wrong setting or delete the wrong resource in the web console you can break things. Infrastructure as code helps solve this, especially when it is combined with version control, such as Git.

Scalability: With infrastructure as code you can write it once and then reuse it many times. This means that one well-written template can be used as the basis for multiple services, in multiple regions around the world, making it much easier to horizontally scale.

Security: Once again infrastructure as code gives you a unified template for how to deploy your architecture. If you create one well-secured architecture you can reuse it multiple times, and know that each deployed version is following the same settings.

Transactional: Terraform not only creates resources on your cloud account but also waits for them to stabilize while they start. It verifies that provisioning was successful, and if there is a failure it throws an error and try to recreate resource again during next apply.

Click here to see a video to know what Infrastructure as Code is, and why is it important?

Click here to know more about Infrastructure as Code

Question 6: **Correct**
Which of the following configuration block has Terraform 0.12 recommended way of provider version constraint?

```
# Terraform Block

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

# Provider Block

provider "aws" {
  version = "~> 3.0"
}
```

- Terraform 0.12 only supports the `required_provider` within terraform block, and `version` is not a valid argument in provider blocks.
- Terraform 0.12 recommend using the `version` in provider blocks, and *not* using the `required_providers` argument within terraform block. `required_providers` is still supported for compatibility with older Terraform versions.
- Terraform 0.12 only supports the `version` within provider blocks, and `required_providers` is not a valid argument in terraform block.
- Terraform 0.12 recommend using the `required_providers` in terraform block, and *not* using the `version` argument within provider blocks. `version` is still supported for compatibility with older Terraform versions.
(Correct)

Explanation

Provider version constraints can also be specified using a `version` argument within a `provider` block, but that simultaneously declares a new provider configuration that may cause problems particularly when writing shared modules. For that reason, we recommend using the `required_providers` block

as described above, and *not* using the `version` argument within `provider` blocks. `version` is still supported for compatibility with older Terraform versions.

```
# Recommend

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

# Only supported for compatibility with older Terraform versions

provider "aws" {
  version = "~> 3.0"
}
```

The version meta-argument specifies a `version` constraint for a provider, and works the same way as the version argument in a `required_providers` block. The `version` constraint in a provider configuration is only used if `required_providers` does not include one for that provider.

Terraform does not recommend using the version argument in provider configurations. In Terraform 0.13 and later, version constraints should always be declared in the required_providers block.

[Click here to know more about Provider Versions]

Note: The `version` meta-argument made sense before ***Terraform 0.13***, since Terraform could only install providers that were distributed by HashiCorp. Now that Terraform can install providers from multiple sources, it makes more sense to keep version constraints and provider source addresses together.

Update: **New in Terraform 0.13**

The `name = { source, version }` syntax for `required_providers` was added in Terraform 0.13. Previous versions of Terraform used a version constraint string instead of an object (like `mycloud = "~> 1.0"`), and had no way to specify provider source addresses. If you want to write a module that works with both Terraform v0.12 and v0.13, see [v0.12-Compatible Provider Requirements] below.

Now, a provider requirement consists of a local name, a source location, and a version constraint:

```
terraform {
  required_providers {
    mycloud = {
      source  = "mycorp/mycloud"
      version = "~> 1.0"
    }
  }
}
```

Each argument in the required_providers block enables one provider. The key determines the provider's local name (its unique identifier within this module), and the value is an object with the following elements:

* `source` - the global source address for the provider you intend to use, such as hashicorp/aws.

* `version` - a version constraint specifying which subset of available provider versions the module is compatible with.

Question 7: **Correct**
You created a Terraform module and pushed it to your company's private git repository. You recently observed that this module has a bug; hence, you checked out a new `hotfix` branch from the `master` branch and fixed it. How can you test the new module code before merging it to the `master`?

-

Run a test configuration by changing the module source to the value given below and make sure it works as expected.

```
1. module "vpc" {
2.   source = "git::https://code.quizexperts.com/vpc.git?branch=hotfix"
3. }
```

- Run a test configuration by changing the module source to the value given below and make sure it works as expected.

```
1. module "vpc" {
2.   source = "git::https://code.quizexperts.com/vpc.git#ref=hotfix"
3. }
```

- Run a test configuration by changing the module source to the value given below and make sure it works as expected.

```
1. module "vpc" {
2.   source  = "git::https://code.quizexperts.com/vpc.git"
3.   branch = "hotfix"
4. }
```

- Run a test configuration by changing the module source to the value given below and make sure it works as expected.

```
1. module "vpc" {
2.   source = "git::https://code.quizexperts.com/vpc.git&branch=hotfix"
3. }
```

- Run a test configuration by changing the module source to the value given below and make sure it works as expected.

```
1. module "vpc" {
2.   source = "git::https://code.quizexperts.com/vpc.git"
3.   version = "hotfix"
4. }
```

- Run a test configuration by changing the module source to the value given below and make sure it works as expected.

```
1. module "vpc" {
2.   source = "git::https://code.quizexperts.com/vpc.git?ref=hotfix"
3. }
```

(Correct)

Explanation

Selecting a Revision

By default, Terraform will clone and use the default branch (referenced by `HEAD`) in the selected repository. You can override this using the `ref` argument:

```
# To specify a Tag

module "vpc" {
  source = "git::https://code.quizexperts.com/vpc.git?ref=v1.2.0"
}


# To specify a Branch

module "vpc" {
  source = "git::https://code.quizexperts.com/vpc.git?ref=hotfix"
}
```

You have a generic ref there that Git will work out what you mean by context assuming no collisions.

If you do have 2 refs that are ambiguous then Git will error and tell you that it's an ambiguous ref and force you to specify the full `ref` using `refs/heads/branch-name` or `refs/tags/tag-name`.

The value of the `ref` argument can be any reference that would be accepted by

the `git checkout` command, including branch and tag names.

Click here to know more about Selecting a revision

Click here to know more about such cases on stackoverflow.com

NOTE: This is an important topic for the exam.

Question 8: **Incorrect**
Which of the following variable type allows multiple values of several distinct types to be grouped together as a single value? Choose *TWO* correct answers.

- Set
- Tuple
(Correct)
- Object
(Correct)
- Map
(Incorrect)
- List

Explanation

Structural Types

A *structural* type allows multiple values of *several distinct types* to be grouped together as a single value. Structural types require a *schema* as an argument, to specify which types are allowed for which elements.

The two kinds of structural type in the Terraform language are:

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

The schema for object types is `{ <KEY> = <TYPE>, <KEY> = <TYPE>, ... }` — a pair of curly braces containing a comma-separated series of `<KEY> = <TYPE>` pairs. Values that match the object type must contain *all* of the specified keys, and the value for each key must match its specified type. (Values with *additional* keys can still match an object type, but the extra attributes are discarded during type conversion.)

For Example: An object type of `object({ name=string, age=number })` would match a value like the following:

```
{
  name = "John"
  age  = 52
}
```

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

The schema for tuple types is `[<TYPE>, <TYPE>, ...]` — a pair of square brackets containing a comma-separated series of types. Values that match the tuple type must have *exactly* the same number of elements (no more and no fewer), and the value in each position must match the specified type for that position.

For Example: A tuple type of `tuple([string, number, bool])` would match a value like the following:

```
["a", 15, true]
```

Click here to know more about Tuple and Objects

Question 9: **Correct**
What command you can use to get human-readable output from a state or plan file?

-

`terraform state`

- `terraform plan`

- `terraform show`
(Correct)

- `terraform workspace list`

Explanation

The `terraform show` command is used to provide human-readable output from a state or plan file. This can be used to inspect a plan to ensure that the planned operations are expected, or to inspect the current state as Terraform sees it.

Click here to know more about show command

Question 10: **Incorrect**
Which of the following statements are true for below given Terraform configuration? Choose *TWO* correct answers.

```
# Create virtual network
resource "azurerm_virtual_network" "vnet" {
    name                = "QuizExpertsVnet"
    address_space       = ["10.0.0.0/16"]
    location            = "westus2"
    resource_group_name = azurerm_resource_group.rg.name
}

## ...more blocks here...

# Create a resource group
resource "azurerm_resource_group" "rg" {
    name     = "QuizExpertsResourceGroup"
    location = "westus2"
}
```

- The order of blocks and expressions in the Terraform configuration file doesn't matter.
(Correct)

- Terraform can infer when one resource depends on another by analyzing the resource attributes used in interpolation expressions. From this, Terraform builds a dependency tree to determine the correct order to create each resource.
(Correct)

- This configuration will fail while creating a new Azure VNet. You have to specify the Resource Group's name to contain the VNet. The value of `resource_group_name` in this configuration is an expression `azurerm_resource_group.rg.name` and will not be available as you are creating Resource Group after Azure VNet.

- If resource blocks and expressions in the configuration are in random order, use the `depends_on` *meta-argument* to define these dependencies explicitly.

Explanation

Implicit dependencies

Terraform can infer when one resource depends on another by analyzing the resource attributes used in interpolation expressions. The act of referencing an attribute from another resource in the argument value of a resource results in an implicit dependency. From this, Terraform builds a dependency tree to determine the correct order in which to create each resource. Use implicit dependencies wherever possible.

In the question of an implicit dependency, Terraform knows that the `azurerm_resource_group` has to be created before

the `azurerm_virtual_network` because the virtual network references the resource group in its `resource_group_name` argument. The expression `azurerm_resource_group.rg.name` creates an implicit dependency on the `azurerm_resource_group` object named `rg`.

```
# Create a resource group
resource "azurerm_resource_group" "rg" {
    name     = "QuizExpertsResourceGroup"
    location = "westus2"
}
# Create virtual network
resource "azurerm_virtual_network" "vnet" {
    name                = "QuizExpertsVnet"
    address_space       = ["10.0.0.0/16"]
    location            = "westus2"
    resource_group_name = azurerm_resource_group.rg.name
}
```

Because of the dependency, the order of blocks and expressions in the configuration file doesn't matter. This example works just like the one above:

```
# Create virtual network
resource "azurerm_virtual_network" "vnet" {
    name                = "QuizExpertsVnet"
    address_space       = ["10.0.0.0/16"]
    location            = "westus2"
    resource_group_name = azurerm_resource_group.rg.name
}

## ...more blocks here...

# Create a resource group
resource "azurerm_resource_group" "rg" {
    name     = "QuizExpertsResourceGroup"
    location = "westus2"
}
```

Explicit dependencies

Sometimes there are dependencies between resources that are not visible to Terraform. In these scenarios, use the `depends_on` *meta-argument* to explicitly define these dependencies. The depends_on meta-argument accepts a list of resources.

Click here to know more about Terraform Resource Dependencies

Question 11: **Incorrect**

To publish a module to Terraform Public Registry. A module must meet the following requirements.

Choose *TWO* correct answers.

- GitHub repository must have HarshiCorp Licence (Version ~> 2).
- The module must be on GitHub and must be a private repo with at least one release tag in x.y.z format.
- The module must be on a public GitHub repo with at least one release tag in x.y.z format.
  (Correct)
- Module GitHub repositories must use this three-part name format **terraform-<PROVIDER>-<NAME>** where <NAME> segment can not contain any special characters.
  (Incorrect)
-

Module GitHub repositories must use this three-part name format **terraform-<PROVIDER>-<NAME>** where <NAME> segment can contain additional hyphens.
(Correct)

- Module GitHub repositories must use this three-part name format **module-<PROVIDER>-<NAME>** where <NAME> segment can contain additional hyphens.
(Incorrect)

Explanation

Requirements

The list below contains all the requirements for publishing a module. Meeting the requirements for publishing a module is extremely easy. The list may appear long only to ensure we're detailed, but adhering to the requirements should happen naturally.

* **GitHub.** The module must be on GitHub and must be a public repo. This is only a requirement for the public registry. If you're using a private registry, you may ignore this requirement.

* **Named `terraform-<PROVIDER>-<NAME>`.** Module repositories must use this three-part name format, where `<NAME>` reflects the type of infrastructure the module manages and `<PROVIDER>` is the main provider where it creates that infrastructure. The `<NAME>` segment can contain additional hyphens. Examples: `terraform-google-vault` or `terraform-aws-ec2-instance`.

* **Repository description.** The GitHub repository description is used to populate the short description of the module. This should be a simple one-sentence description of the module.

* **Standard module structure.** The module must adhere to the standard module structure. This allows the registry to inspect your module and generate documentation, track resource usage, parse submodules and examples, and more.

* **`x.y.z` tags for releases.** The registry uses tags to identify module versions. Release tag names must be a semantic version, which can optionally be prefixed with a `v`. For example, `v1.0.4` and `0.9.2`. To publish a module initially, at least one release tag must be present. Tags that don't look like version numbers are ignored.

Click here to know more about Publishing a Module

Question 12: **Correct**
Refer to the `example.tf` file given below and select the answer that applies. If you run `terraform apply` three times, how many IP records will you see in the file `ip_address.txt`?

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

resource "aws_instance" "quiz_experts" {
  ami           = "ami-05794d4ef61b053f2"
  instance_type = "t2.micro"

  provisioner "local-exec" {
    command = "echo ${aws_instance.quiz_experts.public_ip} >> ip_address.txt"
  }
}
```

- 1
(Correct)

- 3

-

  0

-

  None of these

  Explanation

  Creation-Time Provisioners

  By default, provisioners run when the resource they are defined within is created. Creation-time provisioners are only run during *creation*, not during updating or any other lifecycle. They are meant as a means to perform bootstrapping of a system. Hence no matter how many times you run creation-time (default) provisioners after creating a resourse it will not run.

  ```
  % cat ip_address.txt
  23.20.23.157
  ```

  Click here to know more about Creation-Time Provisioners

Question 13: **Correct**
You delete one or more resources from a Terraform configuration file since the configuration no longer exists. How does Terraform determine what resources it should delete and in which order?

-

  Terraform will no longer manage these resources. We have to delete these manually from the cloud provider management console.

-

  Terraform retains a copy of the most recent set of dependencies within the state. Even if you delete one or more items from the configuration, Terraform can still determine what to delete and the correct order for destruction from the state.
  (Correct)

-

  Since configuration no longer exists, Terraform has no way to delete such resources.

-

  This is still an open issue at Terraform. You should have used `count = 0` parameter instead of removing resources from Terraform configuration, which could help Terraform know that it should remove these resources.

  Explanation

  Terraform requires some sort of database to map Terraform config to the real world. When you have a resource `resource "aws_instance" "quiz_experts"` in your configuration, Terraform uses this map to know that instance `i-abcd1234` is represented by that resource.

  Alongside the mappings between resources and remote objects, Terraform also tracks metadata such as resource dependencies and record it in state files.

  Terraform typically uses the configuration to determine the dependency order. However, when you delete a resource from a Terraform configuration, Terraform must know how to delete that resource. Terraform can see that a mapping exists for a resource, not in your configuration, and plan to destroy. However, since the configuration no longer exists, the order cannot be determined from the configuration alone.

  To ensure correct operation, Terraform retains a copy of the most recent set of dependencies within the state. Now Terraform can still determine the correct order for destruction from the state when you delete one or more items from the configuration.

  Click here to know more about Terraform State

Question 14: **Correct**
Which command automatically updates configurations in the current directory for easy readability and consistency?

-

  `terraform fmt`
  (Correct)

-

  `terraform check`

-

`terraform format`

- `terraform init`

- `terraform validate`

Explanation

Format and validate the configuration

We recommend using consistent formatting in files and modules written by different teams. The `terraform fmt` command automatically updates configurations in the current directory for easy readability and consistency.

Format your configuration. Terraform will return the names of the files it formatted. If your configuration file is already formatted correctly, Terraform won't return any file names.

```
$ terraform fmt
```

Click here to know more

Question 15: **Correct**
A Terraform _____ determines how Terraform loads and stores state.

- Provider

- Resource

- Data Block

- Backend
(Correct)

Explanation

A "**backend**" in Terraform determines how state is loaded and how an operation such as `apply` is executed.

By default, Terraform uses the "local" backend, which stores the state file on your local disk. Remote backends allow you to store the state file in a remote, shared store.

Here are some of 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.

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.

Even if you only intend to use the "local" backend, it may be useful to learn about backends since you can also change the behavior of the local backend.

Click here to know more about Terraform Backends

Question 16: **Correct**
The Terraform Registry is a public registry. For private use of such a browsable marketplace of modules, you must use the Private Registry available in the Terraform Cloud or Enterprise version.

- False

- True
(Correct)

Explanation

If you are a customer of HashiCorp's Terraform Enterprise/ Terraform Cloud, you can have this same experience with a Private Terraform Registry; that is, a registry that lives in your private Git repos, and is only accessible to your team. This can be a great way to share modules within your company.

The Terraform open source project does not provide a server implementation, but community members can create their own private registries by following the published protocol.

Click here to know more about Private Registry

Click here to know more

Question 17: **Correct**
Which of the following is the primitive type constraint in Terraform? Choose *TWO* correct answers.

- Integer
- List
- Array
- String
(Correct)
- Bool
(Correct)

Explanation

Primitive Types

A *primitive* type is a simple type that isn't made from any other type. All primitive types in Terraform are represented by a type keyword. The available primitive types are:

* `string`: a sequence of Unicode characters representing some text, such as `"hello"`.

* `number`: a numeric value. The `number` type can represent both whole numbers like `15` and fractional values such as `6.283185`.

* `bool`: either `true` or `false`. `bool` values can be used in conditional logic.

Click here to know more about Type Constraints in Terraform

Question 18: **Correct**
Fill in the blanks: The _____ environment variable can enable detailed Terraform logs to appear on _____.

- First - TF_CRASH_LOG
Second - stdout
- First - TF_PATH
Second - stderr
- First - TF_LOG
Second - stdout
- First - TF_LOG
Second - stderr
(Correct)

- First - TF_LOGS
  Second - stdout

  Explanation

  Debugging Terraform

  The TF_LOG environment variable can enable detailed Terraform 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.

  Example

  ```
  export TF_LOG=TRACE
  ```

  Click here to know more about debugging in Terraform

  Question 19: **Correct**
  The parent module has to provide input variables and providers to a child module explicitly in the module block as this information can not be passed down to descendent modules implicitly through inheritance.

- True

- False
  (Correct)

  Explanation

  Providers

  - Providers can be passed down to descendent modules in two ways: either *implicitly* through inheritance, or *explicitly* via the `providers` argument within a `module` block. These two options are discussed in more detail in the following sections.

  - Provider configurations can be defined only in a root Terraform module.

  - A module intended to be called by one or more other modules must not contain any provider blocks, with the exception of the special "proxy provider blocks"

  - Additional provider configurations (those with the `alias` argument set) are *never* inherited automatically by child modules, and so must always be passed explicitly using the `providers` map.

  Example:

  ```
  provider "aws" {
    alias  = "usw1"
    region = "us-west-1"
  }

  provider "aws" {
    alias  = "usw2"
    region = "us-west-2"
  }

  module "tunnel" {
    source    = "./tunnel"
    providers = {
      aws.src = aws.usw1
      aws.dst = aws.usw2
    }
  }
  ```

Input Variables

The parent has to provide input variables values to the child module explicitly in the module block.

Example:

```
module "aws_network_vpc" {
  source           = "./modules/vpc"
  aws_vpc_block    = var.aws_vpc_block
  aws_vpc_tag_name = var.aws_vpc_tag_name
}
```

Click here to know more about Providers in Modules

Click here to know more about Calling a Child Module

Click here to know about Proxy Configuration Blocks

Question 20: **Incorrect**
In regards to Terraform List, Select all correct statements. Choose *THREE* correct answers.

- List is a sequence of values identified by consecutive whole numbers starting with zero.
(Correct)
- List is a collection of multiple values of several distinct types grouped together as a single value.
- List is a collection of values identified by a string label.
(Incorrect)
- List is a collection of unique values.
- All elements of a list must always be of the same type.
(Correct)
- `["a", [], "b"]` is a valid Terraform list.
- `["a", 15, true]` is a valid Terraform list.
(Correct)

Explanation

List (**Collection Type**)

A `list` is a sequence of values of the same type identified by consecutive whole numbers starting with zero.

For example, the type `list(string)` means "list of strings", which is a different type than `list(number)`, a list of numbers. All elements of a collection must always be of the same type.

Whenever possible, Terraform converts values between similar kinds of complex types (list/tuple/set and map/object) if the provided value is not the exact type requested, therefore -

* A `tuple` can be converted to a `list`, and vice-versa.
* An `object` can be converted to a `map` and vice-versa.

However, certain conditions should meet to enable this conversion. For examples -

Example 1:

A `list` can only be converted to a `tuple` if it has *exactly* the required number of elements.

Example 2:

If a module argument requires a value of type `list(string)` and a user provides the tuple `["a", 15, true]`, Terraform will internally transform the value to `["a", "15", "true"]` by converting the elements to the required `string` element type as all of the elements of a collection must have the same type.

On the other hand, an automatic conversion will fail if the provided value (including any of its element values) is incompatible with the required type. If an argument requires a type of `list(string)` and a user provides the tuple `["a", [], "b"]` then the value cannot conform to the type constraint, Terraform would reject this value, complaining that all elements must have the same type, since an empty tuple cannot be converted to a string.

NOTE: The keyword `list` is a shorthand for `list(any)`, which accepts any element type as long as every element is the same type. This is for compatibility with older configurations; for new code, we recommend using the full form.

Click here to know more about Terraform list

Click here to know more about Conversion of Complex-types in Terraform

Question 21: **Incorrect**
Which of the following options can enable the most verbose logs and write those logs to a file? Choose *TWO* correct answers.

-
  1. `export TF_LOG="TRACE"`
  2. `export TF_LOG_FILE="terraform.txt"`
-
  1. `export TF_LOG="TRACE"`
  2. `export TF_FILE="terraform.txt"`
-
  1. `export TF_LOG="DEBUG"`
  2. `export TF_LOG_PATH="terraform.txt"`

  (Incorrect)
-
  1. `export TF_LOG="RANDOM"`
  2. `export TF_LOG_PATH="terraform.txt"`

  (Correct)
-
  1. `export TF_LOG="TRACE"`
  2. `export TF_LOG_PATH="terraform.txt"`

  (Correct)

Explanation

Terraform has detailed logs that 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.

Therefore `export TF_LOG="RANDOM"` or `export TF_LOG="TRACE"` both can set the most verbose log level.

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.

```
export TF_LOG_PATH="terraform.txt"
```

Since Terraform writes logs to *stderr* you can also use typical Linux redirection

syntax to write logs into a file, but it is not recommended and you have to add it in every Terraform command.

```
terraform plan 2>> terraform.txt
terraform apply 2>> terraform.txt
```

Click here to know more about TF_LOG and TF_LOG_PATH variables

Question 22: **Correct**
Terraform cannot reason about what the provisioner does; hence if a creation-time provisioner fails for a resource, Terraform will plan to destroy and recreate resource upon the next `terraform apply`.

- True
(Correct)

- False

Explanation

Creation-Time Provisioners

By default, provisioners run when the resource they are defined within is created. Creation-time provisioners are only run during *creation*, not during updating or any other lifecycle. They are meant as a means to perform bootstrapping of a system.

If a creation-time provisioner fails, the resource is marked as **tainted**. A tainted resource will be planned for destruction and recreation upon the next `terraform apply`. Terraform does this because a failed provisioner can leave a resource in a semi-configured state. Because Terraform cannot reason about what the provisioner does, the only way to ensure proper creation of a resource is to recreate it. This is tainting.

You can change this behavior by setting the `on_failure` attribute, which is covered in detail below.

Click here to know more about Creation-Time Provisioner

Question 23: **Correct**
Refer to the configuration block given below and suggest the value of `X` to get `us-east-2` as *output* `selected_region`.

```
variable "region_list" {
  description = "List of AWS regions"
  type        = list(string)
  default     = ["us-east-1", "us-east-2", "us-west-1"]
}

output "selected_region" {
  value = var.region_list[X]
}
```

- None of these

- 1
(Correct)

- 2

- `"us-east-2"`

Explanation

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

Since the list index starts from zero, here are the output values for corresponding X values -

```
variable "region_list" {
```

```
  description = "List of AWS regions"
  type        = list(string)
  default     = ["us-east-1", "us-east-2", "us-west-1"]
}

output "selected_region" {
  value = var.region_list[1]
}
```

Question 24: **Correct**
What does the below-given command do?

153. `export TF_LOG=`

- Enable detailed logs to the file set in environment variable TF_LOG_PATH.
- Does nothing.
- Enable detailed logs to appear on screen.
- Disable detailed logs.
(Correct)

Explanation

To disable, either unset it or set it to empty.

```
# Set it to empty

export TF_LOG=
```

OR

```
# Unset it

unset TF_LOG
```

Click here to know more about TF_LOG

Question 25: **Incorrect**
Sentinel is a _____ service that prevents the provisioning of out-of-policy infrastructure.

- Prescriptive
- Passive
- Reactive
(Incorrect)
- Prescribed
- Proactive
(Correct)

Explanation

Sentinel Policy as Code

Sentinel is a policy as a code framework that's integrated into Hashicorp enterprise products.

Compared to many tools that scan existing infrastructure for policy infractions, Sentinel proactively prevents provisioning of out-of-policy infrastructure. Policy as code can be written to achieve a range of goals from limiting oversized cloud deploys to ensuring all infrastructure is properly tagged.

Sentinel allows users to define policies that are enforced against infrastructure between the plan and apply phases of a Terraform run.

Click here to know more about Sentinel

Question 26: **Incorrect**
In regards to Terraform Cloud, Select all correct statements. Choose *TWO* correct answers.

- Terraform Cloud 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. **(Correct)**
- Terraform Cloud is self-hosted distribution of Terraform. **(Incorrect)**
- Workspaces in open-sourced Terraform and Terraform Cloud are identical.
- Terraform Cloud always encrypts state at rest and protects it with TLS in transit. **(Correct)**

Explanation

Terraform Cloud

Terraform Cloud is an application that helps teams use Terraform together. It manages Terraform runs in a consistent and reliable environment, and includes easy access to shared state and secret data, access controls for approving changes to infrastructure, a private registry for sharing Terraform modules, detailed policy controls for governing the contents of Terraform configurations, and more.

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.

Click here to know more about Terraform Cloud

Question 27: **Correct**
Lois used configuration given below and created three users, "Peter", "Chris", and "Stewie", provided with a `default` value of the `user_names` variable.

```
variable "user_names" {
  description = "Create IAM users"
  type        = list(string)
  default     = ["Peter", "Chris", "Stewie"]
}

resource "aws_iam_user" "quiz_experts_user" {
  count = length(var.user_names)
  name  = var.user_names[count.index]
}
```

What changes will occur if Lois changes `default` value as given below and runs `terraform apply` command?

```
variable "user_names" {
  description = "Create IAM users"
  type        = list(string)
  default     = ["Peter", "Brian"]
}

resource "aws_iam_user" "quiz_experts_user" {
  count = length(var.user_names)
  name  = var.user_names[count.index]
}
```

- Terraform will stop managing user "Chris" and "Stewie" and a new user "Brian"

will get created.

- User "Chris" and "Stewie" will get deleted, and a new user "Brian" will get created.
- User "Chris" and "Stewie" will not get deleted, and a new user "Brian" will get created.
- User "Stewie" will get deleted, and the user "Chris" will get renamed to "Brian". **(Correct)**

Explanation

When you use the `count` parameter on a resource, that resource becomes a list of resources. Unfortunately, the way Terraform identifies each resource within the list is by its position (index) in that list. That is, after running `terraform apply` the first time with three user names, Terraform's internal representation of these IAM users looks something like this:

```
aws_iam_user.quiz_experts_user[0]: Creation complete after 3s [id=Peter]
aws_iam_user.quiz_experts_user[2]: Creation complete after 3s [id=Stewie]
aws_iam_user.quiz_experts_user[1]: Creation complete after 3s [id=Chris]
```

When you remove an item from the list or add new items, all the items after it shifts back or forth, so after running `plan` with just two user names, Terraform's internal representation will look something like this:

```
aws_iam_user.quiz_experts_user[0]: Refreshing state... [id=Peter]
aws_iam_user.quiz_experts_user[1]: Refreshing state... [id=Brian]
```

Complete `terraform apply` command output will look like this:

```
$ terraform apply
aws_iam_user.quiz_experts_user[0]: Refreshing state... [id=Peter]
aws_iam_user.quiz_experts_user[1]: Refreshing state... [id=Chris]
aws_iam_user.quiz_experts_user[2]: Refreshing state... [id=Stewie]

An execution plan has been generated and is shown below.
Resource actions are indicated with the following symbols:
  ~ update in-place
  - destroy

Terraform will perform the following actions:

  # aws_iam_user.quiz_experts_user[1] will be updated in-place
  ~ resource "aws_iam_user" "quiz_experts_user" {
        arn           = "arn:aws:iam::444300417083:user/Chris"
        force_destroy = false
        id            = "Chris"
      ~ name          = "Chris" -> "Brian"
        path          = "/"
        tags          = {}
        unique_id     = "AIDAWO4S6LA5RLSPBMG45"
    }

  # aws_iam_user.quiz_experts_user[2] will be destroyed
  - resource "aws_iam_user" "quiz_experts_user" {
      - arn           = "arn:aws:iam::444300417083:user/Stewie" -> null
      - force_destroy = false -> null
      - id            = "Stewie" -> null
      - name          = "Stewie" -> null
      - path          = "/" -> null
      - tags          = {} -> null
      - unique_id     = "AIDAWO4S6LA53VJPSJCZM" -> null
```

```
    }

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

Do you want to perform these actions?
  Terraform will perform the actions described above.
  Only 'yes' will be accepted to approve.

  Enter a value: yes

aws_iam_user.quiz_experts_user[2]: Destroying... [id=Stewie]
aws_iam_user.quiz_experts_user[1]: Modifying... [id=Chris]
aws_iam_user.quiz_experts_user[2]: Destruction complete after 3s
aws_iam_user.quiz_experts_user[1]: Modifications complete after 4s [id=Brian]

Apply complete! Resources: 0 added, 1 changed, 1 destroyed.
```

Notice how Terraform modifies index 1 user since Terraform sees the index as a resource's identity, to Terraform, this change roughly translates to "*rename the user at index 1 to Brian and delete the user at index 2.*"

Reference

Question 28: **Correct**
The default setting of Terraform Open Source is to store your state file on your local laptop or your workstation.

- False
- True
(Correct)

Explanation

By default, Terraform states are stored in a local file named `terraform.tfstate` under the current working directory, but it can also be stored remotely, which works better in a team environment.

Click here to know more about Terraform State

Question 29: **Incorrect**
Suppose you manage resources such as databases with Terraform similar to the example code given below. Select all security measures you must take to protect sensitive data such as username and password? Choose *THREE* correct answers.

```
resource "aws_db_instance" "quiz_experts_db" {
  engine               = "mysql"
  engine_version       = "5.7"
  instance_class       = "db.t2.micro"
  name                 = "quiz_experts_db"

  # Set the secrets from variables
  username = var.username
  password = var.password
}
```

- If you manage any sensitive data with Terraform (like database passwords, user passwords, or private keys), treat the state itself as sensitive data, always encrypt state at rest, and protect it with TLS in transit.
(Correct)
- Local state provides better security as state data never leaves your system.
- Use *sensitive* argument to stop sensitive values from being recorded in the state.

(Incorrect)

- Strictly control who can access your Terraform backend.
(Correct)

- Storing state remotely can provide better security.
(Correct)

- Use local backend as it always encrypt state files in memory before storing it to files.

Explanation

Sensitive Data in State

Terraform state can contain sensitive data, depending on the resources in use and your definition of "sensitive." The state contains resource IDs and all resource attributes. For resources such as databases, this may contain initial passwords.

When using local state, state is stored in plain-text JSON files.

When using remote state, state is only ever held in memory when used by Terraform. It may be encrypted at rest, but this depends on the specific remote state backend.

Recommendations

If you manage any sensitive data with Terraform (like database passwords, user passwords, or private keys), treat the state itself as sensitive data.

Storing state remotely can provide better security. As of Terraform 0.9, Terraform does not persist state to the local disk when remote state is in use, and some backends can be configured to encrypt the state data at rest.

Click here to know more about Terraform Recommendations for Sensitive Data

Question 30: **Correct**
Connection blocks don't take a block label, and can be nested within either a `resource` or a `provisioner`.

- False

- True
(Correct)

Explanation

Connection blocks don't take a block label, and can be nested within either a `resource` or a `provisioner`.

* A `connection` block nested directly within a `resource` affects all of that resource's provisioners.

```
resource "aws_instance" "quiz_experts" {
  ami           = "ami-04579a6a597"
  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"
    ]
  }
```

```
}
```

* A `connection` block nested in a `provisioner` block only affects that provisioner, and overrides any resource-level connection settings.

```
resource "aws_instance" "quiz_experts" {
  # Something
  # Something

  # Copies the file as the root user using SSH
  provisioner "file" {
    source      = "conf/myapp.conf"
    destination = "/etc/myapp.conf"

    connection {
      type     = "ssh"
      user     = "root"
      password = "${var.root_password}"
      host     = "${var.host}"
    }
  }
}
```

Use case: One use case for providing multiple connections is to have an initial provisioner connect as the `root` user to set up user accounts, and have subsequent provisioners connect as a user with more limited permissions.

Click here to know more about Provisioners Connection

Question 31: **Correct**

Which of the following variable block will produce an error while executing `terraform validate` command?

-
```
variable "cidrs" { default = [] }
```

-
```
variable "amis" {
  type    = map
  default = []
}
```

(Correct)

-
```
variable "node_pools" {
  type        = map(any)
  description = "number of node pools"
}
```

-
```
variable "server_configs" {
  type = object({
    server_ips = list(string)
    username   = string
    password   = string
  })
}
```

Explanation

If both the `type` and `default` arguments are specified, the given default value must be convertible to the specified type. In the example given below -

```
variable "amis" {
  type    = map
  default = []  # This must be a valid map type instead of list.
}
```

NOTE: The keyword `any` maybe used to indicate that any type is acceptable.

[Click here to know more about Variable Type Constraints]

Question 32: **Correct**

You have created a new workspace called `DEV` under directory `/home/quiz_experts/iac/` and initialized terraform by running `terraform init` command with *local* backend. If you run `terraform apply` , where does Terraform will write its state data?

- `/home/quiz_experts/iac/DEV/terraform.tfstate`
- `/home/quiz_experts/iac/terraform.tfstate`
- `/home/quiz_experts/iac/terraform.tfstate.d/DEV/terraform.tfstate` **(Correct)**
- `/home/quiz_experts/iac/workspace.d/terraform.tfstate`
- `/home/quiz_experts/iac/workspace.d/DEV/terraform.tfstate`
- `/home/quiz_experts/iac/terraform.tfstate.d/terraform.tfstate`

Explanation

Workspace

A common use for multiple workspaces is 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. For example, a developer working on a complex set of infrastructure changes might create a new temporary workspace in order to freely experiment with changes without affecting the default workspace.

Workspaces are technically equivalent to renaming your state file. They aren't any more complex than that. Terraform wraps this simple notion with a set of protections and support for remote state.

For local state, Terraform stores the workspace states in a directory called `terraform.tfstate.d`. This directory should be treated similarly to local-only `terraform.tfstate`

So for non-default workspaces state will be stored in a file - `terraform.tfstate.d/<your-workspace-name>/terraform.tfstate`

In this example our current working directory is `/home/quiz_experts/iac/` , hence directory should look like this -

```
.
├── main.tf
├── terraform.tfstate        # Default workspace state file
└── terraform.tfstate.d
    ├── DEV
    |   └── terraform.tfstate   # DEV workspace state file
    └── PROD
        └── terraform.tfstate   # PROD workspace state file
```

For a workspace named `DEV` , state file will be - `/home/quiz_experts/iac/terraform.tfstate.d/DEV/terraform.tfstate`

[Click here to more about workspaces]

Question 33: **Correct**

Peter uses configuration given below to provision an aws_db_instance and uses environment variables to provide values to variables used within the resource block secretly. Can Terraform still expose these sensitive values like passwords in the state file?

```
# ............

variable "db_instance_name" {}
variable "db_username" {}
variable "db_password" {}
```

```
resource "aws_db_instance" "quiz_experts_demo" {
  allocated_storage    = 20
  storage_type         = "gp2"
  engine               = "mysql"
  engine_version       = "5.7"
  instance_class       = "db.t2.micro"
  name                 = var.db_instance_name
  username             = var.db_username
  password             = var.db_password
}

# ............
```

- Yes, all arguments will be stored in the raw state as plain-text; however, you can use sensitive argument to mark a variable containing sensitive value so that Terraform can avoid storing it in plain-text.
- Yes, all arguments, including the username and password, will be stored in the raw state as plain-text.
(Correct)
- No, Terraform does not store sensitive data like username and password in the state file.
- No, Terraform states are encrypted at rest by default.

Explanation

Sensitive Data in State

Terraform state can contain sensitive data, depending on the resources in use and your definition of "sensitive." The state contains resource IDs and all resource attributes. For resources such as databases, this may contain initial passwords.

When using a local state, the state is stored in plain-text JSON files.

Recommendations

If you manage any sensitive data with Terraform (like database passwords, user passwords, or private keys), treat the state itself as sensitive data.

Storing the state remotely can provide better security.

Click here to know more about Sensitive Data in State

Question 34: **Correct**
Peter is writing a module that creates VPC, subnets, security-groups, route-tables, and nat-gateway. Each of these resources has common team-specific tags. Therefore, instead of defining the same tags multiple times in the configuration. What approach should Peter use to make the configuration more readable and easy to update in the future?

- Use Functions
- Use Input Variables
- Use Data Sources
- Use Local Values
(Correct)
- Use Dynamic Blocks

Explanation

When To Use Local Values

Local values can be helpful to avoid repeating the same values or expressions multiple times in a configuration, but if overused they can also make a configuration hard to read by future maintainers by hiding the actual values used.

Use local values only in moderation, in situations where a single value or result is used in many places *and* that value is likely to be changed in the future. The ability to easily change the value in a central place is the key advantage of local values.

Click here to know more about Local Values

Question 35: **Correct**

Your manager has asked you to create a strong separation between `development` and `production` deployments to ensure users will only touch the intended infrastructure. Any change in one deployment should not impact another deployment. Which of the following, you think is the right approach?

- Use separate Terraform workspaces for each deployment as each workspace has a different state file associated with it.
- Modify Terraform configuration and use different resource nomenclature for different deployments.
- Use separate working-directories for each deployment. Each working directory should have its separate deployment-specific configuration and backend with different credentials and access controls.
(Correct)
- Use separate remote backend storage for each deployment.
- Use a deployment name as input variable and create a condition in each resource block to set `count` 0 or 1 based on deployment name.

Explanation

Named workspaces allow conveniently switching between multiple instances of a *single* configuration within its *single* backend. They are convenient in a number of situations, but cannot solve all problems.

A common use for multiple workspaces is 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. For example, a developer working on a complex set of infrastructure changes might create a new temporary workspace in order to freely experiment with changes without affecting the default workspace.

Organizations commonly want to create a strong separation between multiple deployments of the same infrastructure serving different development stages (e.g. staging vs. production) or different internal teams. In this case, the backend used for each deployment often belongs to that deployment, with different credentials and access controls. Named workspaces are *not* a suitable isolation mechanism for this scenario because each subsystem should have its own separate configuration and backend, and will thus have its own distinct set of workspaces.

```
$ tree
.
├── assets
|   ├── index.html
├── prod  # Separate directory for production deployment
|   ├── main.tf
|   ├── variables.tf
|   ├── terraform.tfstate
|   └── terraform.tfvars
└── dev  # Separate directory for development deployment
    ├── main.tf
    ├── variables.tf
    ├── terraform.tfstate
    └── terraform.tfvars
```

Click here to know more

Question 36: **Correct**
Can another process unlock local state files?

- Yes, local state files can be unlocked by another process.
- Local state files cannot be unlocked by another process.
(Correct)

Explanation

The behavior of this lock is dependent on the backend being used. Local state files cannot be unlocked by another process.

Click here to know about force-unlock command

Question 37: **Incorrect**
When it comes to Terraform 0.12, what does the `terraform init` command do? Choose *THREE* correct answers.

- Downloads and installs community as well as HashiCorp distributed provider's plugin so that it can later be executed during terraform plan and apply.
- Searches the configuration for direct and indirect references to providers and load plugins distributed by HashiCorp.
(Correct)
- Searches the configuration and load plugins only if a valid *provider block* configuration is found.
(Incorrect)
- Downloads and initialize backend and provider; however, module code must pre-exist at the locations given in their source arguments.
(Incorrect)
- Sees the root configuration directory for backend configuration, and initialized the chosen backend.
(Correct)
- Searches for module blocks and the source code for referenced modules is retrieved from the locations given in their source arguments.
(Correct)

Explanation

Plugin Installation

During init, Terraform searches the configuration for both direct and indirect references to providers and attempts to load the required plugins.

Backend Initialization

During init, the root configuration directory is consulted for backend configuration and the chosen backend is initialized using the given configuration settings.

Child Module Installation

During init, the configuration is searched for `module` blocks, and the source code for referenced modules is retrieved from the locations given in their `source` arguments.

Community/Third-party Plugins (Terraform 0.12)

Anyone can develop and distribute their own Terraform providers. These third-party providers must be manually installed, since `terraform init` cannot automatically download them.

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.

Providers distributed by HashiCorp can also go in the user plugins directory. If a manually installed version meets the configuration's version constraints, Terraform will use it instead of downloading that provider. This is useful in airgapped environments and when testing pre-release provider builds.

Community/Third-party Plugins (Terraform 0.13) *<- New*

With Terraform 0.13, `terraform init` will automatically download and install partner and community providers in the HashiCorp Terraform Registry, following the same clear workflow as HashiCorp-supported official providers. These improvements to the ecosystem will benefit Terraform users and provider developers alike.

NOTE: Terraform recently announced Terraform 0.13, it can do the automatic installation of third-party (community) providers.

Hence, in the exam, you can be asked about `terraform init` command behavior in Terraform 0.13, then you should also mark -> Downloads and installs community as well as HashiCorp distributed provider's plugin so that it can later be executed during terraform plan and apply.

Click here to know more about the terraform init command

New -> Click here to know Automatic Installation of Third-Party Providers with Terraform 0.13

Question 38: **Correct**
Can the expression of a local value refer to other locals?

- Yes
(Correct)

- No

Explanation

The expression of a local value can refer to other locals, but as usual reference cycles are not allowed. That is, a local cannot refer to itself or to a variable that refers (directly or indirectly) back to it.

Example:

```
locals {
  # Common tags to be assigned to all resources
  common_tags = {
    Service = local.service_name
    Owner   = local.owner
  }
}

resource "aws_instance" "example" {
  # ...

  tags = local.common_tags
}
```

Click here to know more about Locals

Question 39: **Correct**
You created three VMs for an application using a Terraform configuration, as illustrated in this `terraform apply` output:

```
aws_instance.quiz_experts[0]: Creation complete after 42s [id=i-0fe5dc9f90a5c9376]
aws_instance.quiz_experts[1]: Creation complete after 45s [id=i-06240e6517a27559d]
aws_instance.quiz_experts[2]: Creation complete after 42s [id=i-00eb7ce0eed4df485]
```

Now because of some direct manual changes

on `aws_instance.quiz_experts[1]`, you want to re-create just one `aws_instance.quiz_experts[1]` machine. How can you do it?

- Run below given two commands -
  1. `terraform taint "aws_instance.quiz_experts[1]"`
  2. `terraform apply`

  (Correct)

- Run command -
  1. `terraform recreate -target="aws_instance.quiz_experts[1]"`

- Run command -
  1. `terraform taint "aws_instance.quiz_experts[1]"`

- Run below given two commands -
  1. `terraform destroy -auto-approve`
  2. `terraform apply -auto-approve`

- Run below given two commands -
  1. `terraform refresh -target="aws_instance.quiz_experts[1]"`
  2. `terraform apply`

Explanation

Manually Tainting Resources

In cases where you want to manually destroy and recreate a resource, Terraform has a built-in `taint` function in the CLI. 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.

To taint a resource, use the following command:

```
$ terraform taint resource.id
```

`resource.id` refers to the resource block name and resource ID to taint.

```
resource "aws_instance" "quiz_experts_vm" {
  ami           = "ami-fc0a942c"
  instance_type = "t2.micro"
}
```

The correct resource and ID to taint this resource given in the above configuration would be:

```
terraform taint aws_instance.quiz_experts_vm
```

Hence `taint` is the preferred way to do this using built-in re-creation features is to mark the resource as tainted and then to rerun the terraform apply process. Based on `terraform apply` the output given in the question we need to taint *index* 1 instance from the group of instances created by one resource block:

```
terraform taint "aws_instance.quiz_experts[1]"
terraform apply
```

You could also do this by using a targeted terraform destroy:

```
terraform destroy -target="aws_instance.quiz_experts[1]"
terraform apply
```

[Click here to know more about resource taint]

Question 40: **Correct**
How does Infrastructure as Code (IaC) make infrastructure more reliable? Choose *FOUR* correct answers.

- Without IaC, an operator might open multiple sessions during infrastructure

changes, which often results in skipped steps that can result in slight differences between servers that compound over time and could impact their performance, usability, or security.
(Correct)

- Scaling up infrastructure becomes more reliable when the operator manually connects to each machine remotely and provision and configure many servers by executing a series of commands/scripts.

- IaC automates infrastructure changes. It has no relation to its reliability.

- With IaC, we can test the code and review the results before the code is applied to our target environments.
(Correct)

- IaC makes changes idempotent, consistent, repeatable, and predictable.
(Correct)

- Since code is checked into version control systems such as GitHub, GitLab, BitBucket, etc., it is possible to review how the infrastructure evolves over time.
(Correct)

Explanation

IaC Makes Infrastructure More Reliable

IaC makes changes idempotent, consistent, repeatable, and predictable. Without IaC, scaling up infrastructure to meet increased demand may require an operator to remotely connect to each machine and then manually provision and configure many servers by executing a series of commands/scripts. They might open multiple sessions and move between screens, which often results in skipped steps or slight variations between how work is completed, necessitating rollbacks. Perhaps a command was run incorrectly on one instance and reverted before being re-run correctly.

These process inconsistencies can result in slight differences between servers that compound over time and could impact their performance, usability, or security. If a large team is applying changes, the risks increase because individuals don't always follow the same instructions identically.

With IaC, we can test the code and review the results before the code is applied to our target environments. Should a result not align to our expectations, we iterate on the code until the results pass our tests and align to our expectations. Following this pattern allows for the outcome to be predicted before the code is applied to a production environment. Once ready for use, we can then apply that code via automation, at scale, ensuring consistency and repeatability in how it is applied.

Since code is checked into version control systems such as GitHub, GitLab, BitBucket, etc., it is possible to review how the infrastructure evolves over time. The idempotent characteristic provided by IaC tools ensures that, even if the same code is applied multiple times, the result remains the same.

Click here to know more about it

Question 41: **Correct**
A variable is defined by using a *variable* block with a label, which of the following is a valid variable block?

-
```
variable "counts" {
  type        = number
  description = "Number of resources to launch"
}
```
(Correct)

-
```
variable "count" {
  type        = "number"
  description = "Number of resources to launch"
}
```

-
```
variable "count" {
```

```
  type        = number
  description = "Number of resources to launch"
}
```

-

```
variable "counts" {
  type        = integer
  description = "Number of resources to launch"
}
```

Explanation

The label after the `variable` keyword is a name for the variable, which must be unique among all variables in the same module. This name is used to assign a value to the variable from outside and to reference the variable's value from within the module.

The name of a variable can be any valid identifier except the following:

These names are reserved for meta-arguments in module configuration blocks, and cannot be declared as variable names.

The variable declaration can optionally include a `type` argument to specify what value types are accepted for the variable, as described in the following section.

The variable declaration can also include a `default` argument. If present, the variable is considered to be *optional* and the default value will be used if no value is set when calling the module or running Terraform.
The `default` argument requires a literal value and cannot reference other objects in the configuration.

Click here to more about Variables

Question 42: **Correct**
Let's say you are using AWS provider, and you need to create two EC2 instances in two different AWS regions. Is it possible to create resources in more than one region with a single configuration?

-

Yes, we need to define two provider block one for each region as given below -

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

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

-

Yes, region argument under provider blocks supports list.

```
provider "aws" {
  region = [ "us-east-1" , "us-west-2" ]
}
```

-

Yes, include multiple provider blocks with the same provider name. For each additional non-default configuration, use the alias meta-argument to provide a new name segment.
(Correct)

-

Yes, include multiple provider blocks with the different provider name.

-

No, You need to use multiple configurations for the same provider to create resources in different regions.

Explanation

Multiple Provider Configurations

You can optionally define multiple configurations for the same provider, and select which one to use on a per-resource or per-module basis. The primary reason for this is to support multiple regions for a cloud platform; other examples include targeting multiple Docker hosts, multiple Consul hosts, etc.

To create multiple configurations for a given provider, include multiple `provider` blocks with the same provider name. For each additional non-default configuration, use the `alias` meta-argument to provide an extra name segment. For example:

```
# The default provider configuration; resources that begin with `aws_` will use
# it as the default, and it can be referenced as `aws`.
provider "aws" {
  region = "us-east-1"
}

# Additional provider configuration for west coast region; resources can
# reference this as `aws.west`.
provider "aws" {
  alias  = "west"
  region = "us-west-2"
}
```

Selecting Alternate Provider Configurations

By default, resources use a default provider configuration (one without an `alias` argument) inferred from the first word of the resource type name.

To use an alternate provider configuration for a resource or data source, set its `provider` meta-argument to a `<PROVIDER NAME>.<ALIAS>` reference:

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

  # ...
}
```

Click here to know more about Multiple Provider Configurations

Question 43: **Correct**
Terraform recommends using provisioners instead of configuration management tools to invoke scripts or installing software on a newly built VMs.

- False
(Correct)

- True

Explanation

Terraform Provisioners - "*Provisioners are a Last Resort*"

Provisioners should only be used as a last resort. For most common situations there are better alternatives like user_data in AWS EC2 or tools like Chef/Ansible/Saltstack built specifically for configuration management.

Click here to know more

Question 44: **Correct**
Your team uses Terraform to manage dev, staging, and production environments. For every update, you have to update each environment containing duplicate blocks of configuration separately. What Terraform features can you use to avoid this growing burden of configuration updates?

- Modules
(Correct)

- Functions

- Providers

- Packages
- Plugins

Explanation

Terraform Module

As you manage your infrastructure with Terraform, you will create increasingly complex configurations. There is no intrinsic limit to the complexity of a single Terraform configuration file or directory. It is possible to continue writing and updating your configuration files in a single directory. However, if you do, you end up with a very complicated configuration with lots of duplicate blocks. Updating the configuration will become riskier, as an update to one section may cause unintended consequences to other parts of your configuration.

Terraform recommend that every practitioner use modules by following these best practices:

- 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.

- 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 significantly reduce the burden of maintaining and updating your configuration as your infrastructure grows in complexity.

- 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.

- Publish and share modules with your team. Most infrastructure is managed by a team of people, and modules are an important way for teams to work together to create and maintain infrastructure. As mentioned earlier, you can publish modules either publicly or privately.

[Click here to more about Terraform Modules]

[Click here to know more about Module Best Practices]

Question 45: **Correct**

You create a VM with the configuration given below to host an internal tool and push this Terraform configuration to GitHub so that your teammate also can manage the same VM. Your teammate cloned this repository and saw that you named VM as `testing` and changed the resource name to `tool_instance` to make it more relatable with the tool it hosts. Now, how will Terraform behave with VM during the next apply?

***Before*:**

```
terraform {
  backend "s3" {
    bucket         = "quiz-experts"
    key            = "terraform/dev/states"
    region         = "us-east-1"
    dynamodb_table = "terraform-locks"
    encrypt        = true
  }
}

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

resource "aws_instance" "test" {
  ami           = "ami-05794d4ef61b053f2"
  instance_type = "t2.micro"
}
```

After:

```
terraform {
  backend "s3" {
    bucket         = "quiz-experts"
    key            = "terraform/dev/states"
    region         = "us-east-1"
    dynamodb_table = "terraform-locks"
    encrypt        = true
  }
}

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

resource "aws_instance" "tool_instance" {
  ami           = "ami-05794d4ef61b053f2"
  instance_type = "t2.micro"
}
```

- Terraform will do nothing since VM already exists in the state and in the desired state.
- Even if the user has just changed the resource name, Terraform will delete the existing VM and create a new VM with a new name.
(Correct)
- Terraform apply after renaming a resource does not modify infrastructure, but does modify the state file.
- Terraform apply after renaming a resource will create one more resource with the new name, and the user has to delete the old resource manually.

Explanation

Rename a Resource

Terraform tracks resources by their name. If you change the name, you have *created a new resource* and *deleted the old resource*. There is no way for Terraform to tell you changed the name of an existing resource since its all just text. Hence even if the user has just changed the resource name, Terraform will delete the existing VM and create a new VM with a new name.

How to rename a resource instead of deleting it?

Renaming a resource in Terraform without deleting and creating it is a 2 step process.

For example, to rename the `aws_instance` resource named `test` to `tool_instance`:

1) Define your resource block with a new name.

```
# Something

resource "aws_instance" "tool_instance" {
  ami           = "ami-05794d4ef61b053f2"
  instance_type = "t2.micro"
}
```

2) Use the Terraform move command.

```
# Execute the Terraform move command
# by passing your old resource name first, followed by your new resource name.

$ terraform state mv 'aws_instance.test' 'aws_instance.tool_instance'
```

To verify it, run, Terraform plan command after above given two steps, and make sure `terraform plan` says - "*No changes. Infrastructure is up-to-date.*" same as example given below -

```
$ terraform plan

Acquiring state lock. This may take a few moments...

Refreshing Terraform state in-memory prior to plan...

The refreshed state will be used to calculate this plan, but will not be

persisted to local or remote state storage.


aws_instance.tool_instance: Refreshing state... [id=i-018d635d0ab1a2f4b]


------------------------------------------------------------------------


No changes. Infrastructure is up-to-date.


This means that Terraform did not detect any differences between your

configuration and real physical resources that exist. As a result, no

actions need to be performed.
```

Click here to see more resource renaming examples

Question 46: **Correct**

A calling module can directly access child module resource attributes.

- False
(Correct)

- True

Explanation

The resources defined in a module are encapsulated, so the calling module cannot access their attributes directly. However, the child module can declare output values to selectively export certain values to be accessed by the calling module.

For example, if the `./app-cluster` module referenced in the example above exported an output value named `instance_ids` then the calling module can reference that result using the expression `module.instances.instance_ids`:

```
resource "aws_elb" "quiz_experts_elb" {

  # ...


  instances = module.instances.instance_ids

}
```

Click here to know more about Accessing Module Output Values

Question 47: **Correct**

Your company has been using a paid Cloud Management tool to manage resources in AWS. Due to some upper management decisions, the company wants to discontinue this paid Cloud Management tool, and leverage open-source Terraform to move towards Infrastructure as Code and save cost. Now your boss asked you to explore Terraform to manage existing AWS resources. How can it be done without recreating existing resources?

- Add a provider block with valid AWS access key and secrete key to Terraform configuration, and run `terraform refresh` followed by `terraform apply` to manage existing resources with Terraform.

- Existing resources can not be managed by open-source Terraform. You need a paid version to import existing infrastructure directly to the Terraform Cloud state.

-

You have to stop your existing resources, and then use `terraform import` command with `resource-id` to import all resources one by one to Terraform state.

- You can use the `terraform export` command to export existing infrastructure to Terraform's state file, so that Terraform is aware of and can manage that infrastructure.

- You can use the `terraform import` command to add existing infrastructure to Terraform's state file, so that Terraform is aware of and can manage that infrastructure.
(Correct)

Explanation

Terraform Usage

Terraform is able to import existing infrastructure. This allows you to 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.

Currently State Only

The current implementation of Terraform import can only import resources into the state. It does not generate configuration. A future version of Terraform will also generate configuration.

Because of this, prior to running `terraform import` it is necessary to write manually a `resource` configuration block for the resource, to which the imported object will be mapped.

Import Usage

To import a resource, first write a resource block for it in your configuration, establishing the name by which it will be known to Terraform:

```
resource "aws_instance" "quiz_experts_vm" {
  # ...instance configuration...
}
```

The name "quiz_experts_vm" here is local to the module where it is declared and is chosen by the configuration author. This is distinct from any ID issued by the remote system, which may change over time while the resource name remains constant.

If desired, you can leave the body of the resource block blank for now and return to fill it in once the instance is imported.

Now `terraform import` can be run to attach an existing instance to this resource configuration:

```
$ terraform import aws_instance.quiz_experts_vm i-c4gd1563l
```

This command locates the AWS instance with ID `i-c4gd1563l`. Then it attaches the existing settings of the instance, as described by the EC2 API, to the name `aws_instance.quiz_experts_vm` of a module. In this example, the module path implies that the root module is used. Finally, the mapping is saved in the Terraform state.

Click here to know more about terraform import command usage

Question 48: **Correct**
Version constraints are supported only for modules installed from a module registry.

- True
(Correct)

- False

Explanation

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

since they're loaded from the same source repository, they always share the same version as their caller.

Click here to know more about Module Versions

Question 49: **Correct**

You have started getting lots of conflicts and inconsistencies in the infrastructure managed by Terraform ever since You began using Amazon S3 as Terraform backend. After some troubleshooting, you observed that this only happens when more than one user tries to make infrastructure changes with Terraform. How will you resolve it?

- Add rate limiting rule on Amazon S3 bucket to accept only one request for a duration of time.
- Enable Terraform state locking for the Amazon S3 backend by using a DynamoDB table to avoid race around the condition and concurrent execution problem of the same Terraform configuration. **(Correct)**
- Add outbound rate limit in `ip-tables` of the server running terraform CLI.
- Ask users to notify all team members if a user needs to run Terraform CLI so that no other user runs Terraform CLI at the same time.
- Use Terraform workspace as each workspace has a separate state file that resolves conflicts and inconsistencies problem of shared states.
- Open-source Terraform only allows one user to use an S3 backend. You need to upgrade to Terraform Cloud or Enterprise to enable multiple users to work on the same state concurrently.

Explanation

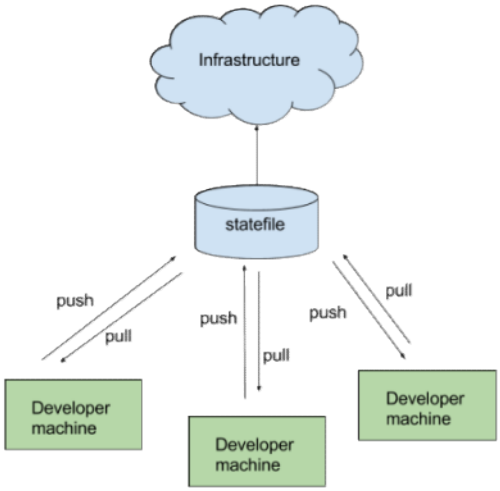


Image source - Internet

Why state locking?

Storing the state remotely lets you collaborate smoothly. It allows each of your team members to access the latest state files, but it also brings a risk of accidental state file corruption, especially when two team members are running Terraform at the same time you may run into race conditions as multiple `terraform apply` commands make concurrent updates to the same state files, leading to conflicts and data loss, and state file corruption.

State lock, a feature that prevents opening a state file while already in use, will resolve this problem. Please note that not all backends support locking. If your backend supports it and is enabled, Terraform will lock your state for all operations that could write state. This prevents others from acquiring the lock and potentially corrupting your state.

NOTE: Not all backends support locking. Please view the list of backend types for details on whether a backend supports locking or not.

You can implement the lock by creating an AWS DynamoDB Table.

Image source - Internet

To use DynamoDB for locking with Terraform, you must create a DynamoDB table with a primary key called `LockID` with this exact spelling and capitalization.

Here's what the sample backend configuration looks like for an S3 backend with state lock:

```
terraform {
  backend "s3" {
    # Replace this with your bucket name.
    bucket         = "quiz-experts-terraform-bucket"
    key            = "global/quiz-experts/stats/terraform.tfstate"
    region         = "us-east-2"

    # Replace this with your DynamoDB table name.
    dynamodb_table = "quiz-experts-terraform-locks"
    encrypt        = true
  }
}
```

Remember to run `terraform init` again after making any change in the remote backend configuration. This command can not only download provider code, but also configure your Terraform backend. Moreover, the init command is *idempotent*, so it's safe to run it over and over again.

Click here to know more about S3 Backend

Click here to know more about State Locking

Question 50: **Correct**

Which of the following describes the right ways of using a local path as module sources? Choose *TWO* correct answers.

-
  1. `module "consul" {`
  2. `  source = "./consul"`
  3. `}`

  (Correct)

-
  1. `module "consul" {`
  2. `  source = "consul"`
  3. `}`

-
  1. `module "consul" {`
  2. `  source = "../consul"`
  3. `}`

  (Correct)

-

```
module "consul" {
  source = "modules/consul"
}
```

Explanation

Local path references allow for factoring out portions of a configuration within a single source repository.

```
module "consul" {
  source = "./consul"
}
```

A local path must begin with either `./` or `../` to indicate that a local path is intended, to distinguish from a module registry address.

Local paths are special in that they are not "installed" in the same sense that other sources are: the files are already present on local disk (possibly as a result of installing a parent module) and so can just be used directly. Their source code is automatically updated if the parent module is upgraded.

Click here to know more about Terraform Module Sources

Question 51: **Correct**
How to add provider to Terraform v0.12 configuration?

-

None of these

-

Both of these
(Correct)

-

Add any resource from the required provider and Terraform will automatically download the provider while running `terraform init` command.

-

Explicitly add provider configuration using a provider block.

Explanation

A new provider can be added to a configuration -- either explicitly via a `provider` block or by adding a resource from that provider -- Terraform must initialize it before it can be used. Initialization downloads and installs the provider's plugin so that it can later be executed.

`terraform init` command will download and initialize any providers that are not already initialized.

NOTE: In Terraform v0.12 `terraform init` command cannot automatically download Third-party Plugins providers that are not distributed by HashiCorp, but in Terraform v0.13 `terraform init` command can do it automatically.

Click here to see the terraform init command

Click here to know more about old the terraform init command

Click here to see what is new in Terraform v0.13

Question 52: **Correct**
You are working on a new load balancer module, and your boss asked you to leverage an already created instance module written by a different team. How can you source this module from a separate private git repository?

-

```
module "instances" {
  source = "ssh://username@code.quizexperts.com:instances.git"
}
```

-

```
module "instances" {
  source = "git::ssh://username@code.quizexperts.com/instances.git"
}
```

(Correct)

-

```
module "instances" {
  source = "git@code.quizexperts.com/terraform/instances.git"
}
```

-

```
module "instances" {
  source = "https://code.quizexperts.com/instances.git"
}
```

Explanation

Generic Git Repository

Arbitrary Git repositories can be used by prefixing the address with the special `git::` prefix. After this prefix, any valid Git URL can be specified to select one of the protocols supported by Git.

For example, to use HTTPS or SSH:

```
module "vpc" {
  source = "git::https://example.com/vpc.git"
}

module "storage" {
  source = "git::ssh://username@example.com/storage.git"
}
```

Terraform installs modules from Git repositories by running `git clone`, and so it will respect any local Git configuration set on your system, including credentials. To access a non-public Git repository, configure Git with suitable credentials for that repository.

If you use the SSH protocol then any configured SSH keys will be used automatically. This is the most common way to access non-public Git repositories from automated systems because it is easy to configure and allows access to private repositories without interactive prompts.

If using the HTTP/HTTPS protocol, or any other protocol that uses username/password credentials, configure Git Credentials Storage to select a suitable source of credentials for your environment.

If your Terraform configuration will be used within Terraform Cloud, only SSH key authentication is supported, and keys can be configured on a per-workspace basis.

Click here to know more about sourcing modules from a private git repo

GitHub

Terraform will recognize unprefixed `github.com` URLs and interpret them automatically as Git repository sources.

```
module "consul" {
  source = "github.com/hashicorp/example"
}
```

The above address scheme will clone over HTTPS. To clone over SSH, use the following form:

```
module "consul" {
  source = "git@github.com:hashicorp/example.git"
}
```

Click here to know more about sourcing modules from GitHub

Question 53: **Correct**

The `terraform import` command currently can only import one resource at a time.

-

True
(Correct)

-

False

Explanation

The `terraform import` command currently can only import one resource at a time. This means you can't yet point Terraform import to an entire collection of resources such as an AWS VPC and import all of it. This workflow will be improved in a future version of Terraform.

Warning: Terraform expects that each remote object it is managing will be bound to only one resource address, which is normally guaranteed by Terraform itself having created all objects. If you import existing objects into Terraform, be careful to import each remote object to only one Terraform resource address. If you import the same object multiple times, Terraform may exhibit unwanted behavior. For more information on this assumption, see the State section.

Click here to know more about terraform import command

Question 54: **Incorrect**
Peter Griffin creates two workspaces named dev and prod.

Select all statements that apply to the context. Choose *THREE* correct answers.

- Peter will have three workspaces named dev, prod, and default. **(Correct)**
- Each workspace will point to the same state file `terraform.tfstate`.
- Peter can delete dev workspace, but he can not delete the default workspace. **(Correct)**
- State from each workspace will point to the same resources in the real world.
- Each workspace has a separate state file associated with it. **(Correct)**
- Peter will have two workspaces named dev and prod.

Explanation

Initially, the backend has only one workspace, called "default", and thus there is only one Terraform state associated with that configuration.

Terraform starts with a single workspace named "default". This workspace is special both because it is the default and also because it cannot ever be deleted. If you've never explicitly used workspaces, then you've only ever worked on the "default" workspace.

Certain backends support *multiple* named workspaces, allowing multiple states to be associated with a single configuration. The configuration still has only one backend, but multiple distinct instances of that configuration to be deployed without configuring a new backend or changing authentication credentials.

When you create a new workspace, Terraform will not see any existing resources that existed on the default (or any other) workspace. **These resources still physically exist,** but are managed in another Terraform workspace.

For local state, Terraform stores the workspace states in a directory called `terraform.tfstate.d`. This directory should be treated similarly to local-only `terraform.tfstate` .

Click here to know more about Terraform Workspaces

Question 55: Skipped
Which of the following represents a valid source string to use a module from a private registry?

```
module "vpc" {
  source = "_________________________________"
  version = "0.9.3"
}
```

- <NAMESPACE>/<PROVIDER>/<NAME>
- <HOSTNAME>/<NAME>/<PROVIDER>
- <HOSTNAME>/<NAMESPACE>/<NAME>/<PROVIDER>
(Correct)
- <NAMESPACE>/<NAME>/<PROVIDER>
- <HOSTNAME>/<NAMESPACE>/<PROVIDER>/<NAME>

Explanation

Private Registry Module Sources

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.

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

Click here to know more about Private Registry Module Sources

Question 56: Skipped
Remote state allows teams to share infrastructure resources in a read-only way without relying on any additional configuration store.

- True
(Correct)
- False

Explanation

Delegation and Teamwork with Remote State

Remote state gives you more than just easier version control and safer storage. It also allows you to delegate the outputs to other teams. This allows your infrastructure to be more easily broken down into components that multiple teams can access.

Put another way, remote state also allows teams to share infrastructure resources in a read-only way without relying on any additional configuration store.

For example, a core infrastructure team can handle building the core machines, networking, etc. and can expose some information to other teams to run their own infrastructure. As a more specific example with AWS: you can expose things such as VPC IDs, subnets, NAT instance IDs, etc. through remote state and have other Terraform states consume that.

For example usage, see the `terraform_remote_state` data source.

While remote state is a convenient, built-in mechanism for sharing data between configurations, it is also possible to use more general stores to pass settings both to other configurations and to other consumers. For example, if your environment has HashiCorp Consul then you can have one Terraform configuration that writes to Consul using `consul_key_prefix` and then another that consumes those values using the `consul_keys` data source.

Click here to know more

Question 57: Skipped
Every Terraform configuration has at least one module.

-

True
(Correct)

- False

Explanation

Every Terraform configuration has at least one module, known as its *root module*, which consists of the resources defined in the `.tf` files in the main working directory.

Click here to know more about Terraform Modules

Fullscreen

Question: 1

Terraform init can indeed be run only a few times, because, every time terraform init will initialize the project , and download all plugins from the internet repository , regardless of whether they were present or not , and this increases the waiting times

A. True

B. False

Question: 2

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

A. False

B. True

Question: 3

Which of the below configuration file formats are supported by Terraform? (Select TWO)

A. Node

B. JSON

C. Go

D. YAML

E. HCL

Question: 4

Please identify the offerings which are unique to Terraform Enterprise, and not available in either Terraform OSS, or Terraform Cloud. Select four.

A. Audit Logs

B. Private Network Connectivity

C. VCS Integration

D. Sentinel

E. Clustering

Question: 5

terraform refresh command will not modify infrastructure but does modify the state file.

A. True

B. False

Question: 6

Which of the following clouds does not have a provider maintained HashiCorp

A. IBM Cloud

B. Digital Ocean

C. OpenStack

D. AWS

Question: 7

You have declared a variable name my_var in terraform configuration without a value associated

with it.

variable my_var {}

After running terraform plan it will show an error as variable is not defined.

A. True

B. False

Question: 8

Environment variables can be used to set variables. The environment variables must be in the format "____"_<variablename>. Select the correct prefix string from the following list.

A. TF_CLI_ARGS

B. TF_VAR

C. TF_VAR_

D. TF_VAR_ENV

Question: 9

The terraform init 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.

A. False

B. True

Question: 10

ABC Enterprise has recently tied up with multiple small organizations for exchanging database

information. Due to this, the firewall rules are increasing and are more than 100 rules. This is leading

firewall configuration file that is difficult to manage. What is the way this type of configuration can be managed easily?

A. Terraform Backends

B. Terraform Functions

C. Dynamic Blocks

D. Terraform Expression

Question: 11

True or False: A list(...) contain a number of values of the same type while an object(...) can contain a number of values of different types.

A. False

B. True

Question: 12

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 the arguments are omitted, we call this a _____.

A. First Time Configuration

B. Default Configuration

C. Changing Configuration

D. Partial Configuration

E. Incomplete Configuration

Question: 13

lookup retrieves the value of a single element from which of the below data type?

A. map

B. set

C. string

D. list

Question: 14

Terraform must track metadata such as resource dependencies. Where is this data stored?

A. workspace

B. backend

C. state file

D. metadata store

Question: 15

Which of the below are paid features of Terraform Cloud?

A. Full API Coverage

B. Secure variable Storage

C. Roles/ Team management

D. Cost Estimation

E. Private Module Registry

F. Sentinel policies

Question: 16

Provisioners should only be used as a last resort.

A. False

B. True

Question: 17

Which Terraform command will force a marked resource to be destroyed and recreated on the next

apply?

A. terraform fmt

B. terraform destroy

C. terraform taint

D. terraform refresh

Question: 18

What allows you to conveniently switch between multiple instances of a single configuration within its single backend?

A. Local backends

B. Providers

C. Remote backends

D. Workspaces

Question: 19

Terraform refresh will update the state file?

A. True

B. False

Question: 20

What does terraform plan do?

A. Create an execution plan by evaluating the difference between configuration file and state file.

B. Performs a refresh, unless explicitly disabled, and then apply the changes that are necessary to achieve the desired state specified in the configuration files.

C. Create an execution plan by evaluating the difference between configuration file and

actual infrastructure.

D. Checks whether the execution plan for a set of changes matches your expectations by making changes to real resources or to the state.

Question: 21

When TF_LOG_PATH is set, TF_LOG must be set in order for any logging to be enabled.

A. False

B. True

Question: 22

Workspaces in Terraform provides similar functionality in the open source, Terraform Cloud, and Enterprise versions of Terraform.

A. True

B. False

Question: 23

The current implementation of Terraform import can only import resources into the state. It does not generate configuration.

A. False

B. True

Question: 24

Terraform has detailed logs which can be enabled by setting the _____ environmental variable.

A. TF_TRACE

B. TF_DEBUG

C. TF_LOG

D. TF_INFO

Question: 25

Which of the following best describes a Terraform provider?

A. A plugin that Terraform uses to translate the API interactions with the service or provider.

B. Serves as a parameter for a Terraform module that allows a module to be customized.

C. Describes an infrastructure object, such as a virtual network, compute instance, or other components.

D. A container for multiple resources that are used together.

Question: 26

Which one of the following will run echo 0 and echo 1 on a newly created host?

A. provisioner "local-exec" {command = "echo 0" command = "echo 1"}

B. provisioner "remote-exec" {

```
        inline = [
                echo 0,
                echo 1
                ]
        }
```

C. provisioner "remote-exec" {

```
                command = "${echo 0}"
                command = "${echo 1}"
                }
```

D. **provisioner "remote-exec" {**

```
                inline = [
                        "echo 0",
                        "echo 1"
                        ]
                }
```

Question: 27

How can you ensure that the engineering team who has access to git repo will not create

any non-compliant resources that might lead to a security audit failure in future. your team is using Hashicorp Terraform Enterprise Edition.

A. Use Terraform OSS Sentinel Lite version, which will save cost, since there is no charge for OSS, but it can still check for most non-compliant rules using Policy-As-Code.

B. Implement a review process where every code will be reviewed before merging to the master

branch.

C. Since your team is using Hashicorp Terraform Enterprise Editon, enable Sentinel, and write

Policy-As-Code rules that will check for non-compliant resource provisioning, and prevent/report

them.

D. Create a design /security document (in PDF) and share to the team, and ask them to always follow that document, and never deviate from it.

Question: 28

If you enable TF_LOG = DEBUG, the log will be stored in syslog.log file in the current directory.

A. False

B. True

Question: 29

When using remote state, state is only ever held in memory when used by Terraform.

A. False

B. True

Question: 30

Which one of the following command will rewrite Terraform configuration files to a canonical format

and style.

A. terraform graph -h

B. terraform init

C. terraform graph

D. terraform fmt

Question: 31

You have created a custom variable definition file testing.tfvars. How will you use it for provisioning infrastructure?

A. terraform apply -var-state-file ="testing.tfvars"

B. terraform plan -var-file="testing.tfvars"

C. terraform apply -var-file="testing.tfvars"

D. terraform apply var-file="testing.tfvars"

Question: 32

You want to get involved in the development of Terraform. As this is an open source project, you would like to contribute a fix for an open issue of Terraform. What programming language will need to use to write the fix?

A. It depends on which command issue related to.

B. Python

C. Go

D. HTML

Question: 33

Terraform import command can import resources into modules as well directly into the root of your

state.

A. True

B. False

Question: 34

Which of the following type of variable allows multiple values of several distinct types to be grouped together as a single value?

A. Map

B. Object

C. Tuple

D. List

Question: 35

Which of the following best describes the default local backend?

A. The local backend is where Terraform Enterprise stores logs to be processed by an log collector.

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

C. The local backend is the directory where resources deployed by Terraform have direct access to in order to update their current state.

D. The local backend is how Terraform connects to public cloud services, such as AWS, Azure, or GCP.

Question: 36

Which of the following command can be used to view the specified version constraints for all providers used in the current configuration.

A. terraform providers

B. terraform state show

C. terraform provider

D. terraform plan

Question: 37

Which of the following represents a feature of Terraform Cloud that is NOT free to customers?

A. Roles and Team Management

B. Workspace Management

C. Private Module Registry

D. VCS Integration

Question: 38

What is the purpose of using the local-exec provisioner? (Select Two)

A. To invoke a local executable.

B. Executes a command on the resource to invoke an update to the Terraform state.

C. To execute one or more commands on the machine running Terraform.

D. Ensures that the resource is only executed in the local infrastructure where Terraform is deployed.

Question: 39

terraform state subcommands such as list are read-only commands, do read-only commands create state backup files?

A. Yes

B. No

Question: 40

You want to use terraform import to start managing infrastructure that was not originally provisioned through infrastructure as code. Before you can import the resource's current state, what must you do in order to prepare to manage these resources using Terraform?

A. Run terraform refresh to ensure that the state file has the latest information for existing resources.

B. Update the configuration file to include the new resources.

C. Shut down or stop using the resources being imported so no changes are inadvertently missed.

D. Modify the Terraform state file to add the new resources.

Question: 41

Which of the below terraform commands do not run terraform refresh implicitly before taking actual action of the command?

A. terraform apply

B. terraform destroy

C. terraform init

D. terraform import

E. terraform plan

Question: 42

You want to use different AMI images for different regions and for the purpose you have defined following code block.

```
variable "images"
{
type = "map"

default = {
us-east-1 = "image-1234"
us-west-2 = "image-4567"
us-west-1 = "image-4589"
}
}
```

What of the following approaches needs to be followed in order to select image-4589?

A. var.images["us-west-1"]

B. var.images[3]

C. var.images[2]

D. lookup(var.images["us-west-1"]

Question: 43

By default, a defined provisioner is a creation-times provisioner.

A. True

B. False

Question: 44

What is the default backend for Terraform?

A. consul

B. gcs

C. local

D. etcd

Question: 45

You want terraform plan and apply to be executed in Terraform Cloud's run environment, but the output is to be streamed locally. Which one of the below you will choose?

A. Local Backends

B. This can be done using any of the local or remote backends

C. Remote Backends

D. Terraform Backends

Question: 46

Which of the following Terraform files should be ignored by Git when committing code to a repo? (select Three)

A. Files named exactly terraform.tfvars or terraform.tfvars.json.

B. Any files with names ending in. auto.tfvars or .auto.tfvars.json.

C. input.t

D. terraform.tfstate

E. output.tf

Question: 47

What is the command you can use to set an environment variable named "var1"of type String?

A. export TF_VAR_VAR1

B. set TF_VAR_var1

C. variable "var1" { type = "string"}

D. export TF_VAR_var1

Question: 48

Terraform works well in Windows but a Windows server is required.

A. False

B. True

Question: 49

You have created 2 workspaces PROD and RQ

A. You have switched to RQA and provisioned RQA infrastructure from this workspace. Where is your state file stored?

A. terraform.tfstate.d

B. terraform.d

C. terraform.tfstate.RQA

D. terraform.tfstate

Question: 50

Refer to the below code where developer is outputting the value of the database password but has used sensitive parameter to hide the output value in the CLI.

```
output "db_password" {
    value    = aws_db_instance.db.password
    description = "The password for logging in to the database."
    Sensitive = true
    }
```

Since sensitive is set to true, the value associated with db password will not be present in state file as plain-text?

A. False

B. True

Question: 51

While using generic git repository as a module source, which of the below options allows terraform to select a specific version or tag instead of selecting the HEAD.

A. Append ref argument as

module "vpc" { source = "git::https://example.com/vpc.git?ref=v1.2.0"}

B. Append version argument as

module "vpc" { source = "git::https://example.com/vpc.git?version=v1.2.0"}

C. Append ref argument as

module "vpc" { source = "git::https://example.com/vpc.git#ref=v1.2.0"}

D. By default, Terraform will clone and use the default branch (referenced by HEAD) in the selected repository and you cannot override this.

Question: 52

In regard to deploying resources in multi-cloud environments, what are some of the benefits of using Terraform rather than a provider's native tooling? (select three)

A. Terraform can help businesses deploy applications on multiple clouds and on-premises infrastructure.

B. Terraform is not cloud-agnostic and can be used to deploy resources across a single public cloud.

C. Terraform simplifies management and orchestration, helping operators build large-scale, multi-

cloud infrastructure.

D. Terraform can manage cross-cloud dependencies.

Question: 53

Matt wants to import a manually created EC2 instance into terraform so that he can manage the EC2 instance through terraform going forward. He has written the configuration file of the EC2 instance before importing it to Terraform. Following is the code:

```
resource  "aws_instance"  "mat_ec2"    {
        ami     = "ami-bg2640de"
        instance_type     = "t2.micro"
         vpc_security_group_ids = ["sg-6ae7d613", "sg-53370035"]
        key_name = "mysecret"
        subnet_id = "subnet-9e3cfc5"
        }
```

The instance id of that EC2 instance is i-0260835eb7e9bd40 How he can import data of EC2 to state

file?

A. terraform import aws_instance.id = i-0260835eb7e9bd40

B. terraform import i-0260835eb7e9bd40

C. terraform import aws_instance.i-0260835eb7e9bd40

D. terraform import aws_instance.mat_ec2 i-0260835eb7e9bd40

Question: 54

John wants to use two different regions to deploy two different EC2 instances. He has specified two provider blocks in his providers.t file.

```
provider "aws" { region = "us-east-1" }
provider "aws" { region = "us-west-2" }
```

When he run terraform plan he encountered an error. How to fix this?

A. Use another provider version

B. Use alias for region = "us-west-2"

C. Use default keyword with region = "us-east-1"

D. It cannot be fixed

Question: 55

How does Terraform handle working with so many providers?

A. Terraform ships with all of the plugins embedded in the Terraform binary.

B. Terraform uses a plugin architecture for providers and only installs the provider plugins required by your configuration in the configuration's working directory.

C. Terraform uses a plugin architecture for providers and only installs the provider plugins required by your configuration in a shared, system-wide plugins directory.

D. Terraform allows you to select the providers you want to support during the Terraform installation process.

Question: 56

What is the standard workflow that a developer follows while working with terraform open source

version?

A. Run terraform refresh to update the terraform state , then write the terraform code , and finally run terraform apply.

B. Run terraform destroy first since you need to start from fresh every times , before running terraform apply.

C. Write terraform code , and run terraform push , to update the terraform state to the remote repo , which in turn will take care of the next steps.

D. Write the terraform code on the developer machine, run terraform plan to check the changes , and run terraform apply to provision the infra.

Question: 57

You are using a terraform operation that writes state. Unfortunately, automatic state unlocking has failed for that operation. Which of the below commands can be used to remove the already acquired lock on the state?

A. terraform unlock

B. terraform force-unlock

C. terraform state unlock

D. None of the above

Question: 58

Ric wants to enable detail logging and he wants highest verbosity of logs. Which of the following environment variable settings is correct option for him to select.

A. Set TF_LOG = DEBUG

B. Set VAR_TF = TRACE

C. Set TF_LOG = TRACE

D. Set VAR_TF_LOG = TRACE

Question: 59

The canonical format may change in minor ways between Terraform versions, so after upgrading Terraform it is recommended to proactively run.

A. terraform fmt

B. terraform init

C. terraform validate

D. terraform plan

Question: 60

What does terraform refresh command do?

A. A. terraform refresh can be used to selectively update sections of the state file, using terraform resource level addressing.

B. terraform refresh command basically updates the configuration file with the current state of the actual infrastructure

C. terraform refresh is used to change/modify the infrastructure based on the existing state file, at that

moment.

D. terraform refresh can be used to selectively update sections of the state file, using terraform resource level addressing.

E. terraform refresh syncs the state file with the real-world infrastructure.

Question 1: Skipped
Consider the configuration provided below and select the output if a user uses the file terraform.tfvars given below to provide value to the variable user_details.

```
# Terraform Configuration

variable "user_details" {
```

```
type = object({ name = string, age = number })
}

output "user_details_output" {
value = var.user_details
}
```

```
# Content of terraform.tfvars file

user_details = {
name = "Peter Griffin"
age  = 46
city = "Quahog"
show = "Family Guy"
}
```

```
user_details_output = {
"age"  = 46
"city" = "Quahog"
"name" = "Peter Griffin"
"show" = "Family Guy"
}
```

```
user_details_output = {
"age" = 46
"name" = "Peter Griffin"
}
```
(Correct)

```
user_details_output = {
"age"  : 46,
"name" : "Peter Griffin"
}
```

Error: Invalid value for input variable

Explanation

A map (or a larger object) can be converted to an object if it has *at least* the keys required by the object schema. Any additional attributes are discarded during conversion, which means map -> object -> map conversions can be lossy.

```
user_details_output = {
  "age" = 46
  "name" = "Peter Griffin"
}
```

Click here to know more about conversion of complex-types

Question 2: Skipped
In a Terraform configuration, it's possible to have multiple levels of nested modules. Which of the following command can taint a VM named salt_master within a module named instance that exists under another module named salt_master_cluster?

terraform taint module.salt_master_cluster.module.instance.aws_instance.salt_master
(Correct)

terraform taint module.salt_master_cluster.module.instance.salt_master

terraform taint salt_master_cluster.instance.salt_master

terraform taint module.salt_master_cluster.instance.aws_instance.salt_master

terraform taint salt_master_cluster.instance.aws_instance.salt_master

Explanation

The taint command can be used to taint specific resources within a module:

```
terraform taint module.salt_master_cluster.module.instance.aws_instance.salt_master
```

Click here to know more about Tainting

Question 3: Skipped
Provisioners should only be used as a last resort.

False

True
(Correct)

Explanation

Terraform Provisioners - "*Provisioners are a Last Resort*"

Provisioners should only be used as a last resort. For most common situations there are better alternatives like user_data in AWS EC2 or tools like Chef/Ansible/Saltstack built specifically for configuration management.

Click here to know more

Question 4: Skipped
The terraform _____ command prints information about the providers used in the current configuration.

- providers
  (Correct)
- versions
- init
- status

Explanation

The terraform providers command prints information about the providers used in the current configuration.

This command gives an overview of all of the current dependencies, as an aid to understanding why a particular provider is needed.

```
terraform providers
.
├── provider.aws
├── provider.azurerm
├── provider.google
└── provider.vmc
```

Click here to know more about terraform providers command

Question 5: Skipped
For local state, Terraform stores the workspace states in a directory called _____ .

- terraform.tfstate.d
  (Correct)
- terraform.d/workspace
- terraform.d
- workspace

Explanation

Workspace

For local state, Terraform stores the workspace states in a directory called terraform.tfstate.d. This directory should be treated similarly to local-only terraform.tfstate.

Click here to know more about Terraform Workspace

Question 6: Skipped
Which command can be used to verify whether a configuration files in a directory are syntactically valid and internally consistent?

- terraform state
- terraform validate
  (Correct)
- terraform fmt
- terraform show

Explanation

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.

It is safe to run this command automatically, for example as a post-save check in a text editor or as a test step for a re-usable module in a CI system.

Validation requires an initialized working directory with any referenced plugins and modules installed. To initialize a working directory for validation without accessing any configured remote backend, use:

```
$ terraform init -backend=false
```
If dir is not specified, then the current directory will be used.

To verify configuration in the context of a particular run (a particular target workspace, input variable values, etc), use the terraform plan command instead, which includes an implied validation check.

Click here to know more

Question 7: Skipped
When using remote state, state is only ever held in memory when used by Terraform.

- Ture
(Correct)
- False

Explanation

Terraform state can contain sensitive data, hence while using remote state, Terraform does not persist state to the local disk; instead, state is only ever held in memory when used by Terraform.

Click here to know more

Question 8: Skipped
By default, a provisioner defined in a resource is a _____ provisioner.

- end-tiem
- exec-on
- exec-always
- destroy-time
- start-time
- creation-time
(Correct)

Explanation

By default, a provisioner defined in a resource is a creation-time provisioner.

If when = destroy is specified, the provisioner will run when the resource it is defined within is *destroyed*.

Creation-Time Provisioners

By default, provisioners run when the resource they are defined within is created. Creation-time provisioners are only run during *creation*, not during updating or any other lifecycle. They are meant as a means to perform bootstrapping of a system.

If a creation-time provisioner fails, the resource is marked as **tainted**. A tainted resource will be planned for destruction and recreation upon the next terraform apply. Terraform does this because a failed provisioner can leave a resource in a semi-configured state. Because Terraform cannot reason about what the provisioner does, the only way to ensure proper creation of a resource is to recreate it. This is tainting.

Destroy-Time Provisioners

If when = destroy is specified, the provisioner will run when the resource it is defined within is *destroyed*.

```
resource "aws_instance" "web" {
  # ...
provisioner "local-exec" {
    when    = destroy
    command = "echo 'Destroy-time provisioner'"
  }
}
```

Destroy provisioners are run before the resource is destroyed. If they fail, Terraform will error and rerun the provisioners again on the next terraform apply. Due to this behavior, care should be taken for destroy provisioners to be safe to run multiple times.

Destroy-time provisioners can only run if they remain in the configuration at the time a resource is destroyed.

Click here to know more about Creation-Time Provisioners

Click here to know more about Destroy-Time Provisioners

Question 9: Skipped
What does terraform plan command do?

- Creates an execution plan and then determines what actions are necessary to achieve the desired state by evaluating the difference between the configuration file and real infrastructure.
- Checks whether the execution plan for a set of changes matches your expectations by making necessary changes to real resources or to the state.
- Creates an execution plan and then determines what actions are necessary to achieve the desired state by evaluating the difference between the configuration file and state file.
(Correct)
- Terraform performs a refresh, unless explicitly disabled, and then applies changes necessary to achieve the desired state specified in the configuration files.

Explanation

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.

This command is a convenient way to check whether the execution plan for a set of changes matches your expectations without making any changes to real resources or to the state. For example, terraform plan might be run before committing a change to version control, to create confidence that it will behave as expected.

Click here to know more about plan command

Question 10: Skipped
Select all the correct statements about using modules over a monolithic configuration. Choose *FOUR* correct answers.

- Modules provide consistency in your configurations, which can slow down deployment when you have frequent infrastructure updates.
- Allows you to reuse a piece of proven, tested, documented infrastructure.
(Correct)
- Allows you to share a piece of configuration you have written with your team or the general public, giving them the benefit of your hard work.
(Correct)
- Modules may not be ideal for multi-cloud deployments.
- Modules help to provide consistency in your configurations and make it easier to understand.
(Correct)
- Modules enable you to version your configuration. So production might be using version 1 of a module code, whereas dev and staging could be using version 2 or 3, depending on what level of testing you've done and how ready it is for production.
(Correct)

Explanation

Terraform modules solve the monolith problem

Instead of using a giant, monolithic repo, you can convert that repository into what we call a "modular repo," or smaller, bite-size Terraform configs. Terraform supports the concept of modules, or reusable bits of Terraform code that are small and replaceable. This way, if you need to make a change to part of your infrastructure, you just change the module that you're working on, instead of having to worry about changing the whole monolithic repo.

Let's go through an example. Suppose I have an application that includes a few Linux servers, a load balancer, maybe a DNS record, some security rules, networking—all of the bits and pieces that make up your application stack. What I could do is take the networking code out of my Terraform config and move it over into its own module. And that way, that module can be reused over and over again inside of my code. But if I need to change it, it lives in its own directory, in its own workspace, and I don't have to change the entire mono repo just to make a small change to my module.

Another advantage of modules is that you can version them. So production might be using version 1 of my module code, whereas dev and staging could be using version 2 or 3, depending on what level of testing you've done and how ready that code is for production.

Getting from mono to modules

You might be asking yourself, "How do I go from this big, monolithic repository to a smaller, module-based repository?" The answer is, you don't have to do it all at once. You can break it down into steps. Take part of your monorepo, and take out the parts that make sense to manage separately, like the network, for example. Or you might manage the load balancer inside of its own module.

Gradually you can break that monorepo down into a bunch of smaller repositories that could be managed by different individuals or different teams. This makes it much easier to manage your code, because you're not dealing with one gigantic codebase. You have lots of smaller codebases that you can work on independently.

Click here to watch video from a Solutions Engineer at HashiCorp on Decompose Monolithic Terraform Configurations

Question 11: Skipped
Can you run scripts inside a provisioner block without being attached to any "real" resource?

- No, provisioners can be defined only within a real resource. Terraform error outs if you define provisioner out of resource.
- Yes, by defining provisioners on a null_resource, you can run your scripts as part of the Terraform life cycle without being attached to any "real" resource.
(Correct)
- Yes, you can define provisioner without resource block to run a command outside resource scope.

Explanation

PROVISIONERS WITH NULL_RESOURCE

Provisioners can be defined only within a resource. Still, sometimes, you want to execute a provisioner without tying it to a specific resource. You can do this using something called the null_resource, which acts like a regular Terraform resource, except that it doesn't create anything. By defining provisioners on the null_resource, you can run your scripts as part of the Terraform life cycle without being attached to any "real" resource.

```
resource "null_resource" "quiz-experts" {
  provisioner "local-exec" {
    command = "echo \"Hello, $(whoami)\""
  }
}
```

The null_resource even has a useful argument termed triggers, which takes a map of keys and values. Whenever the values change, the null_resource will be recreated, forcing any provisioners within it to be re-executed. For instance, if you want to execute a provisioner within a null_resource every time you run terraform apply command, you can use the uuid() built-in function, which returns a new, randomly generated UUID each time.

```
resource "null_resource" "quiz-experts" {
  triggers = {
    uuid = uuid()
  }
  provisioner "local-exec" {
    command = "echo \"Hello, $(whoami)\""
  }
}
```

Reference: Book Terraform Up & Running: Writing Infrastructure as Code

Question 12: Skipped
Terraform can be used to codify the configuration for software-defined networks.

- True
  (Correct)
- False

Explanation

Terraform can be used to codify the configuration for software-defined networks. This configuration can then be used by Terraform to automatically setup and modify settings by interfacing with the control layer. This allows configuration to be versioned and changes to be automated. As an example, AWS VPC is one of the most commonly used SDN implementations, and can be configured by Terraform.

Click here to know more about it

Question 13: Skipped
Refer to the configuration block given below and answer the following questions.

- How many users will it create?

- What are the names of the users created by this configuration?

```
resource "aws_iam_user" "quiz_experts_user" {
count = 2
name  = "user_${count.index}"
}
```

- 3, user_1, user_2, user_3
- 3, user_0, user_1, user_2
- 2, user_1, user_2
- 2, user_0, user_1
  (Correct)

Explanation

How many users will it create? - *Two*

- Like Python or any other language, Terraform does not have loops; use the count argument to provision multiple resources with a single resource block.

- The count argument replicates the given resource or module a specific number of times with an incrementing counter. It works best when resources will be identical, or nearly so.

- Use count.index to get the index of each iteration.

- If the count argument is two, it will create two replicas of the same resource block.

What are the names of the users created by this configuration? - *"user_0"* and *"user_1"*

If the count argument is two, count.index will start from zero and end at one with a total of two different values. Hence it will create *"user_0"* and *"user_1"*

```
resource "aws_iam_user" "quiz_experts_user" {
  count = 2
  name  = "user_${count.index}"
}

An execution plan has been generated and is shown below.
Resource actions are indicated with the following symbols:
  + create
```

```
Terraform will perform the following actions:

# aws_iam_user.quiz_experts_user[0] will be created
  + resource "aws_iam_user" "quiz_experts_user" {
      + arn          = (known after apply)
      + force_destroy = false
      + id           = (known after apply)
      + name         = "user_0"
      + path         = "/"
      + unique_id    = (known after apply)
    }

# aws_iam_user.quiz_experts_user[1] will be created
  + resource "aws_iam_user" "quiz_experts_user" {
      + arn          = (known after apply)
      + force_destroy = false
      + id           = (known after apply)
      + name         = "user_1"
      + path         = "/"
      + unique_id    = (known after apply)
    }

Plan: 2 to add, 0 to change, 0 to destroy.
```

Click here to see more examples on Count

Question 14: Skipped

How does Terraform treat a resource if it was successfully created but failed during provisioning? Choose *THREE* correct answers.

- Resource failed during provisioning has been physically created but can't be considered safe to use; hence Terraform will remove this resource and will create new again during next apply.
  (Correct)
- Terraform automatically rolls back and destroys the resource during the apply when the failure happens, because staled resources would go against the execution plan.
- Resource failed during provisioning has been physically created but can't be considered safe to use; hence Terraform will re-run provisioner on this resource during next apply.
- Terraform will error and will delete the resource automatically.
- Terraform will error and mark the resource as "staled".
- Terraform does not automatically roll back and destroy the resource during the apply when the failure happens, because that would go against the execution plan.
  (Correct)
- As resource has been created hence Terraform will not error and continue to create remaining resources.
- Terraform will error and mark the resource as "tainted".
  (Correct)

Explanation

Failed Provisioners and Tainted Resources

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.

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.

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.

Click here to know more about failed provisioners and tainted resources

Question 15: Skipped

Chris Griffin created a Terraform configuration given below to create a security group to allow ports as provided in the configuration. The configuration looks unnecessary long. What Terraform features you can suggest to him to avoid repeatable nested blocks.

```
resource "aws_vpc" "quiz_experts" {
#.....
}

resource "aws_security_group" "allow_access_demo" {
name       = "allow_tls"
description = "Allow TLS inbound traffic"
vpc_id      = aws_vpc.quiz_experts.id

ingress {
from_port   = 443
to_port     = 443
protocol    = "tcp"
cidr_blocks = [aws_vpc.quiz_experts.cidr_block]
}
ingress {
from_port   = 80
```

```
59. to_port     = 80
60. protocol    = "tcp"
61. cidr_blocks = [aws_vpc.quiz_experts.cidr_block]
62. }
63. ingress {
64. from_port   = 8080
65. to_port     = 8080
66. protocol    = "tcp"
67. cidr_blocks = [aws_vpc.quiz_experts.cidr_block]
68. }
69. ingress {
70. from_port   = 8000
71. to_port     = 8000
72. protocol    = "tcp"
73. cidr_blocks = [aws_vpc.quiz_experts.cidr_block]
74. }
75.
76. egress {
77. from_port   = 0
78. to_port     = 0
79. protocol    = "-1"
80. cidr_blocks = ["0.0.0.0/0"]
81. }
82.
83. tags = {
84. Name = "Quiz Experts"
85. Task = "Demo"
86. }
87. }
```

- Local Values
- Functions
- Input Variables
- Dynamic Blocks
  (Correct)

Explanation

Dynamic Blocks

Dynamic Blocks allow us to dynamically construct repeatable nested blocks.

A dynamic block acts much like a for loop.

Here is how we can modify Terraform configuration given in the question -

```
variable "ingress_ports" {
  type        = list(number)
  description = "list of ingress ports"
  default     = [443, 80, 8080, 8000]
}
resource "aws_vpc" "quiz_experts" {
  cidr_block = "172.16.0.0/16"
  tags = {
    Name = "Quiz Experts Demo"
  }
}

resource "aws_security_group" "allow_access_demo" {
  name        = "allow_tls"
  description = "Allow TLS inbound traffic"
  vpc_id      = aws_vpc.quiz_experts.id

dynamic "ingress" {
    iterator = port
    for_each = var.ingress_ports
    content {
      from_port   = port.value
      to_port     = port.value
      protocol    = "tcp"
      cidr_blocks = [aws_vpc.quiz_experts.cidr_block]
    }
  }

egress {
    from_port   = 0
    to_port     = 0
    protocol    = "-1"
    cidr_blocks = ["0.0.0.0/0"]
  }

tags = {
    Name = "Quiz Experts"
    Task = "Demo"
  }
}
```

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.

Click here to know more about Dynamic Blocks

Question 16: Skipped
Which of the following command is used to list all resources within a Terraform state?

-

- terraform list
- terraform list all
- terraform state list
  (Correct)
- terraform state
- terraform show

Explanation

The terraform state list command is used to list resources within a Terraform state.

Example: All Resources

This example will list all resources:

```
$ terraform state list
aws_instance.foo
aws_instance.bar[0]
aws_instance.bar[1]
module.elb.aws_elb.main
```

Example: Filtering by Resource

This example will only list resources for the given name:

```
$ terraform state list aws_instance.bar
aws_instance.bar[0]
aws_instance.bar[1]
```

Example: Filtering by Module

This example will only list resources in the given module:

```
$ terraform state list module.elb
module.elb.aws_elb.main
```

Click here to know more about terraform state list command

Question 17: Skipped
If a backend supports, state locking happens automatically on all write operations on a state.

- False
- True
  (Correct)

Explanation

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

If acquiring the lock is taking longer than expected, Terraform will output a status message. If Terraform doesn't output a message, state locking is still occurring if your backend supports it.

Not all backends support locking. Please view the list of backend types for details on whether a backend supports locking or not.

Click here to know more about Terraform State Lock

Question 18: Skipped
Refer to the Terraform configuration provided at the end and suggest the value of output magic_number for the given terraform.auto.tfvars and terraform.tfvars files.

```
# File terraform.auto.tfvars

input_number = 2
```

```
# File terraform.tfvars

input_number = 3
```

```
# Terraform Configuration

variable "input_number" {
type    = number
default = 1
}

output "magic_number" {
value = var.input_number
}
```

- 1
- 2
  (Correct)
- Error
- 3

Explanation

Variable Definition Precedence

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.)

Click here to know more about variable definition precedence

Question 19: Skipped

Read the statements given carefully and choose the correct options that define the purpose of the Terraform state. Choose *THREE* correct answers.

- Terraform stores a cache of the attribute values for all resources in the state for performance improvement.
  (Correct)
- Terraform states is used to retrieve crash logs of the last `terraform apply` command.
- Terraform states can be used to make infrastructure changes if you do not want to install a Terraform binary on your system.
- Terraform uses the state to track metadata, such as resource dependencies.
  (Correct)
- Terraform uses the state to map configuration to resources in the real world.
  (Correct)

Explanation

Purpose of Terraform State

State is a necessary requirement for Terraform to function. Here are the reasons -

Mapping to the Real World

Terraform requires some sort of database to map Terraform config to the real world. When you have a resource `resource "aws_instance" "foo"` in your configuration, Terraform uses this map to know that instance `i-abcd1234` is represented by that resource.

For some providers like AWS, Terraform could theoretically use something like AWS tags. Early prototypes of Terraform actually had no state files and used this method. However, we quickly ran into problems. The first major issue was a simple one: not all resources support tags, and not all cloud providers support tags.

Therefore, for mapping configuration to resources in the real world, Terraform uses its own state structure.

Metadata

Alongside the mappings between resources and remote objects, Terraform must also track metadata, such as resource dependencies.

Terraform typically uses the configuration to determine the dependency order. However, when you delete a resource from a Terraform configuration, Terraform must know how to delete that resource. Terraform can see that a mapping exists for a resource, not in your configuration, and plan to destroy. However, since the configuration no longer exists, the order cannot be determined from the configuration alone.

To ensure correct operation, Terraform retains a copy of the most recent set of dependencies within the state. Now Terraform can still determine the correct order for destruction from the state when you delete one or more items from the configuration.

One way to avoid this would be for Terraform to know a required ordering between resource types. For example, Terraform could know that servers must be deleted before the subnets they are a part of. The complexity for this approach quickly explodes, however: in addition to Terraform having to understand the ordering semantics of every resource for every cloud, Terraform must also understand the ordering *across providers*.

Terraform also stores other metadata for similar reasons, such as a pointer to the provider configuration that was most recently used with the resource in situations where multiple aliased providers are present.

Performance

In addition to basic mapping, Terraform stores a cache of the attribute values for all resources in the state. This is the most optional feature of the Terraform state and is done only as a performance improvement.

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 small infrastructures, Terraform can query your providers and sync the latest attributes from all your resources. This is the default behavior of Terraform: for every plan and apply; Terraform will sync all resources in your state.

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. On top of this, cloud providers almost always have API rate limiting, so Terraform can only request a certain number of resources in a period of time. Larger users of Terraform make heavy use of the -refresh=false flag as well as the -target flag in order to work around this. In these scenarios, the cached state is treated as the record of truth.

Syncing

In the default configuration, Terraform stores the state in a file in the current working directory where Terraform was run. This is okay for getting started, but when using Terraform in a team, it is important for everyone to be working with the same state so that operations will be applied to the same remote objects.

Remote state is the recommended solution to this problem. With a fully-featured state backend, Terraform can use remote locking as a measure to avoid two or more different users accidentally running Terraform at the same time, and thus ensure that each Terraform run begins with the most recent updated state.

Click here to see official Terraform document

Question 20: Skipped
Which of the following statements are true for below given Terraform configuration? Choose *TWO* correct answers.

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

resource "aws_instance" "quiz_experts_ec2" {
ami          = "ami-b374d5a5"
instance_type = "t2.micro"
}

resource "aws_eip" "quiz_experts_eip" {
vpc      = true
instance = aws_instance.quiz_experts_ec2.id
}
```

- Terraform will create the Elastic IP address before creating the EC2 instance due to the interpolation expression that passes the ID of the EC2 instance to the Elastic IP address.
- Terraform will create the EC2 instance before creating the Elastic IP address due to the interpolation expression that passes the ID of the EC2 instance to the Elastic IP address. **(Correct)**
- In the example above, dependencies between resources are *not* visible to Terraform. We can use depends_on to declare the dependency *explicitly*.
- Terraform will create EC2 instance and Elastic IP address concurrently to reduce the total time taken to apply change.
- In the example above, the reference to aws_instance.quiz_experts_ec2.id creates an *implicit dependency* on the aws_instance named quiz_experts_ec2. **(Correct)**
- depends_on meta-argument is missing in Elastic IP resource block hence Terraform has no way to find dependencies between resources and will try to create these resources parallelly and eventually throw an error as id attribute from the aws_instance.quiz_experts_ec2 resource won't be known until the aws_instance is created.

Explanation

Terraform waited until the creation of the EC2 instance quiz_experts_ec2 was complete before creating the Elastic IP address.

Terraform automatically infers when one resource depends on another by studying the resource attributes used in interpolation expressions. In the example above, the reference to aws_instance.quiz_experts_ec2.id in the definition of the aws_eip.quiz_experts_eip block creates an *implicit dependency*.

Terraform uses this dependency information to determine the correct order in which to create the different resources. To do so, it creates a dependency graph of all of the resources defined by the configuration. In the example above, Terraform knows that the

EC2 Instance must be created before the Elastic IP.

Click here to know more about Terraform Dependencies

Question 21: Skipped
To manage manually created infrastructure resources with Terraform, you must first write a resource block for it in your configuration, establishing the name by which it will be known to Terraform.

- False
- True
(Correct)

Explanation

Import Usage

To import a resource, first write a resource block for it in your configuration, establishing the name by which it will be known to Terraform:

```
resource "aws_instance" "quiz_experts_vm" {
  # ...instance configuration...
}
```

The name "quiz_experts_vm" here is local to the module where it is declared and is chosen by the configuration author. This is distinct from any ID issued by the remote system, which may change over time while the resource name remains constant.

If desired, you can leave the body of the resource block blank for now and return to fill it in once the instance is imported.

Now terraform import can be run to attach an existing instance to this resource configuration:

```
$ terraform import aws_instance.quiz_experts_vm i-c4gd1563l
```

This command locates the AWS instance with ID i-c4gd1563l. Then it attaches the existing settings of the instance, as described by the EC2 API, to the name aws_instance.quiz_experts_vm of a module. In this example, the module path implies that the root module is used. Finally, the mapping is saved in the Terraform state.

Click here to know more about terraform import command usage

Question 22: Skipped
All input variables of the root-module are available to child-module by default.

- False
(Correct)
- True

Explanation

The parent has to provide input variables values to the child module explicitly in the module block.

To understand the question, let's consider an example where we pass variables from the parent to the child modules.

* Parent calls module aws_network_vpc and passes values aws_vpc_block and aws_vpc_tag_name :

```
module "aws_network_vpc" {
  source           = "./modules/vpc"
  aws_vpc_block    = var.aws_vpc_block
  aws_vpc_tag_name = var.aws_vpc_tag_name
}
```

* Module accepts values from calling parent:

```
variable "aws_vpc_tag_name" {
  description = "VPC Name"
}
variable "aws_vpc_block" {
  description = "CIDR for the VPC"
}
```

* Actual module uses vars:

```
resource "aws_vpc" "default" {
  cidr_block           = var.aws_vpc_block
  enable_dns_support   = true
  enable_dns_hostnames = true
tags = {
    Name = var.aws_vpc_tag_name
  }
}
```

Click here to know more about Calling a Child Module

Question 23: Skipped
Which of the following Terraform function will result in an error for the below-given

value of variable ami_id?

Function 1: lookup(var.ami_id, 'prod')

Function 2: lookup(var.ami_id, "stage", "ami-xyz")

```
> var.ami_id
{
"dev" = "ami-456"
"prod" = "ami-123"
}
```

- None of these
- Function 1
  (Correct)
- Both of these
- Function 2

Explanation

Function 1 Output :

```
> lookup(var.ami_id, 'prod')
> Error: Invalid character

on <console-input> line 1:
  (source code not available)

Single quotes are not valid. Use double quotes (") to enclose strings.

Error: Invalid expression

on <console-input> line 1:
  (source code not available)

Expected the start of an expression, but found an invalid expression token.
```

The right way to define the function 1 is lookup(var.ami_id, "prod")

Function 2 Output :

```
> lookup(var.ami_id, "stage", "ami-xyz")
ami-xyz
```

Click here to know more about reading map

Question 24: Skipped
Is it compulsory to provide a version constraint string to pull module from the Terraform registry?

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

- True
- False
  (Correct)

Explanation

source - Require meta-argument

version - Optional meta-arguments

Click here to know more about Other Meta-arguments supported in Module Block

Question 25: Skipped
Terraform forces every state modification command to write a backup file.

- True
  (Correct)
- False

Explanation

State Backups

All terraform state subcommands that modify the state write backup files. The path of these backup file can be controlled with -backup.

Subcommands that are read-only (such as list) do not write any backup files since they aren't modifying the state.

Note that backups for state modification *can not be disabled*. Due to the sensitivity of the state file, Terraform forces every state modification command to write a backup file. You'll have to remove these files manually if you don't want to keep them around.

Click here to know more

Question 26: Skipped
Declaring a variable without any value or type associated with it will cause an error at run time as the variable is not defined.

127. **variable "variable_name" {}**

- False
(Correct)
- True

Explanation

Input variables are usually defined by stating a *name*, *type*, and *default* value. However, the type and default values are optional. Terraform can deduct the type of the variable from the default or input value.

Variables can be predetermined in a file or included in the command-line options. As such, the simplest variable is just a name if there is no value defined, then Terraform prompts the user to provide its value while running commands like plan or apply.

```
variable "variable_name" {}
```

Click here to know more about assigning values to variables

Question 27: Skipped
Each workspace in Terraform Cloud retains backups of its previous state files.

- False
- True
(Correct)

Explanation

Each workspace in Terraform Cloud retains backups of its previous state files. Although only the current state is necessary for managing resources, the state history can be useful for tracking changes over time or recovering from problems

Click here to know more about Terraform Cloud

Question 28: Skipped
Select all the correct statements about providers below. Choose *THREE* correct answers.

- A child module automatically inherits default (un-aliased) provider configurations from its parent.
(Correct)
- A child module does not automatically inherit provider configurations from its parent.
- Each module must declare its own provider requirements so that Terraform can ensure that there is a single version of the provider that is compatible with all modules in the configuration.
(Correct)
- The providers determines where the state is stored.
- Providers are responsible for implementing Policy as a Service before resource creation to ensure all resources have required Tags.
- Provider configurations can be defined only in a root Terraform module.
(Correct)

Explanation

Providers Within Modules

In a configuration with multiple modules, there are some special considerations for how resources are associated with provider configurations.

Each resource in the configuration must be associated with one provider configuration. Provider configurations, unlike most other concepts in Terraform, are global to an entire Terraform configuration and can be shared across module boundaries. Provider configurations can be defined only in a root Terraform module.

Providers can be passed down to descendent modules in two ways: either *implicitly* through inheritance, or *explicitly* via the providers argument within a module block.

Provider Version Constraints in Modules

Although provider *configurations* are shared between modules, each module must declare its own provider requirements, so that Terraform can ensure that there is a single version of the provider that is compatible with all modules in the configuration and to specify the source address that serves as the global (module-agnostic) identifier for a provider.

To declare that a module requires particular versions of a specific provider, use a required_providers block inside a terraform block:

```
terraform {
  required_providers {
    aws = {
      source  = "hashicorp/aws"
```

```
    version = ">= 2.7.0"
  }
 }
}
```

A provider requirement says, for example, "This module requires version v2.7.0 of the provider hashicorp/aws and will refer to it as aws." It doesn't, however, specify any of the configuration settings that determine what remote endpoints the provider will access, such as an AWS region; configuration settings come from provider *configurations*, and a particular overall Terraform configuration can potentially have several different configurations for the same provider.

Click here to know more about it

Question 29: Skipped

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

- True
  (Correct)
- False

Explanation

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

With IaC, we can test the code and review the results before the code is applied to our target environments. Should a result not align to our expectations, we iterate on the code until the results pass our tests and align to our expectations. Following this pattern allows for the outcome to be predicted before the code is applied to a production environment. Once ready for use, we can then apply that code via automation, at scale, ensuring consistency and repeatability in how it is applied.

Since code is checked into version control systems such as GitHub, GitLab, BitBucket, etc., it is possible to review how the infrastructure evolves over time. The idempotent characteristic provided by IaC tools ensures that, even if the same code is applied multiple times, the result remains the same.

Click here to know more about Infrastructure as Code

Question 30: Skipped

An output containing confidential data like a password can be marked as sensitive.

```
output "db_password" {
value       = aws_db_instance.db.password
description = "The password for logging in to the database."
sensitive   = true
}
```

Since the sensitive argument is set to true, can the value associated with the password still be available in plain-text in the state file for everyone to read?

- No, Terraform automatically encrypts outputs containing 'pass', 'secret', or 'key' string in name.
- Yes, Setting an output value in the root module as sensitive prevents Terraform from showing its value in the list of outputs at the end of terraform apply . Sensitive output values are still recorded in the state, and so will be visible to anyone who is able to access the state data.
  (Correct)
- No, Setting an output value in the root module as sensitive prevents Terraform from showing its value in the list of outputs at the end plus only encrypted value goes to state files.
- Yes, However *sensitive* argument is only available in Terraform Cloud Workspaces. Marking output as sensitive has no effect in Terraform OSS.

Explanation

Output can be marked as containing sensitive material using the optional sensitive argument.

Setting an output value in the root module as sensitive prevents Terraform from showing its value in the list of outputs at the end of terraform apply. It might still be shown in the CLI output for other reasons, like if the value is referenced in an expression for a resource argument.

Sensitive output values are still recorded in the state, and so will be visible to anyone who is able to access the state data.

Click here to know more about handling sensitive data in Terraform

Question 31: Skipped

The current implementation of Terraform import can only import resources into the state. It does not generate configuration.

- True
  (Correct)
- False

Explanation

Currently State Only

The current implementation of Terraform import can only import resources into the state. It does not generate configuration. A future version of Terraform will also generate configuration.

Because of this, prior to running terraform import it is necessary to write manually a resource configuration block for the resource, to which the imported object will be mapped.

Click here to know more about it

Question 32: Skipped
Each Terraform configuration has an associated _____ that defines how operations are executed and where persistent data such as the Terraform state are stored.

- backend
  (Correct)
- provider
- remote state
- plugin
- workspace

Explanation

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

Click here to read official document

Click here to read more about Terraform Backend

Question 33: Skipped
As shown in the configuration below. You use an archive stored in Amazon S3 as a module source, which has been updated recently. However, even after running terraform init, you do not see the updated module code under terraform/modules/ directory. In order to solve it, What should you do next?

```
module "vpc" {
source = "s3::https://s3-eu-west-1.amazonaws.com/quiz-experts-s3/vpc.zip"
}
```

- Use new version SHA hash with versionId parameter in the https URL.

```
module "vpc" {
source = "s3::https://s3-eu-west-1.amazonaws.com/quiz-experts-s3/vpc.zip?
versionId=SHAofNewVersion"
}
```

- This happens when user accidentally marks the object (module archive) private. Make it public and rerun the command terraform init.
- Use new version SHA hash as value for version meta-argument.

```
module "vpc" {
source  = "s3::https://s3-eu-west-1.amazonaws.com/quiz-experts-s3/vpc.zip"
version = "new_verion_id"
}
```

- By default terraform init command will not upgrade an already-installed module; use the -upgrade option to upgrade to the newest available version.
  terraform init -upgrade
  (Correct)
- Check network connectivity to S3 by manually running aws s3 ls command and try terraform init again.
- Check user permission on the S3 bucket.

Explanation

By default terraform init command will not upgrade an already-installed module; use the -upgrade option to instead upgrade to the newest available version.

Click here to know more about it

Question 34: Skipped
Which of the following action is not performed by terraform apply command?

- Resource Modification
- Import Resource
  (Correct)
- Destroy Resource
- Provision Resource

Explanation

The terraform apply command is used to apply the changes required to reach

the *desired* state of the configuration. Hence, to reach the desired state, terraform apply command may delete, modify, or provision a new resource. However, the terraform apply command will never import a manually created resource without updating Terraform configuration and terraform import command.

Click here to know more about terraform apply command

Click here to know more about terraform import command

Question 35: Skipped
It is good to define the provider version in the Terraform configuration, operator intended to work with, to avoid accidental upgrades to incompatible new versions. Select all acceptable versions given in the options that meet the provided constraint. Choose *TWO* correct answers.

Terraform v0.12 and earlier

```
terraform {
required_providers {
aws = "~> 0.8.4"
}
}
```

- 1.0.0
- 0.8.6
  (Correct)
- 0.8.4
  (Correct)
- 0.9

Explanation

Version Constraint Syntax

~> 0.8.4 is equivalent to >= 0.8.4, < 0.9

The following operators are valid:

* = (or no operator): Allows only one exact version number. Cannot be combined with other conditions.

* != : Excludes an exact version number.

* >, >=, <, <=: Comparisons against a specified version, allowing versions for which the comparison is true. "Greater-than" requests newer versions, and "less-than" requests older versions.

* ~>: Allows the specified version, plus newer versions that only increase the *most specific* segment of the specified version number. For example, ~> 0.9 is equivalent to >= 0.9, < 1.0, and ~> 0.8.4, is equivalent to >= 0.8.4, < 0.9. This is usually called the pessimistic constraint operator.

UPDATE: Since Terraform recently released **v0.13**. Provide version constraints are slightly changed and Terraform has added one new argument source .

New syntax for Terraform v0.13 and later

```
terraform {
  required_providers {
    mycloud = {
      source  = "hashicorp/aws"
      version = "~> 0.8.4"
    }
  }
}
```

Old syntax for Terraform v0.12 and earlier

```
terraform {
  required_providers {
    aws = "~> 0.8.4"
  }
}
```

Click here to know more about version constraints

Question 36: Skipped
Refer to the configuration given below and select the statement that applies.

```
terraform {
required_providers {
aws = {
source  = "hashicorp/aws"
version = "3.1.0"
}
}
}

```

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

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

- This is not a valid configuration that Terraform supports.
- This is a valid configuration that Terraform 0.12 and above supports.
  (Correct)

Explanation

The terraform {} block is required so Terraform knows which provider to download from the Terraform Registry. In the configuration above, the aws provider's source is defined as hashicorp/aws which is shorthand for registry.terraform.io/hashicorp/aws.

You can also assign a version to each provider defined in the required_providers block. The version argument is optional, but recommended. It is used to constrain the provider to a specific version or a range of versions in order to prevent downloading a new provider that may possibly contain breaking changes. If the version isn't specified, Terraform will automatically download the most recent provider during initialization.

The required_providers block must be nested inside the top-level terraform block (which can also contain other settings).

Each argument in the required_providers block enables one provider. The key determines the provider's local name (its unique identifier within this module), and the value is an object with the following elements:

* source - the global source address for the provider you intend to use, such as hashicorp/aws.

* version - a version constraint specifying which subset of available provider versions the module is compatible with.

NOTE: The name = { source, version } syntax for required_providers was added in Terraform v0.13. Previous versions of Terraform used a version constraint string instead of an object (like mycloud = "~> 1.0"), and had no way to specify provider source addresses. If you want to write a module that works with both Terraform v0.12 and v0.13, see v0.12-Compatible Provider Requirements below.

Question 37: Skipped
Which command can be used to inspect the current state?

- terraform state
- terraform view
- terraform plan
- terraform show
  (Correct)

Explanation

The terraform show command is used to provide human-readable output from a state or plan file. This can be used to inspect a plan to ensure that the planned operations are expected, or to inspect the current state as Terraform sees it.

Click here to know more about show command

Question 38: Skipped
As given below, Glenn has created a variable of *string* type. Select all accepted input from the options given below.

```
variable "my_variable" {
type = string
}
```

- 15
- None of these
- Both of these
  (Correct)
- "15"

Explanation

The Terraform language will automatically convert number and bool values to string values when needed, and vice-versa as long as the string contains a valid representation of a number or boolean value.

* true converts to "true", and vice-versa

* false converts to "false", and vice-versa

* 15 converts to "15", and vice-versa

Click here to know more about type conversion in Terraform

Question 39: Skipped
Can we create multiple instances of a module from a single module block defined in your Terraform configuration? Choose *TWO* correct answers.

- Yes, declare multiple module instances using the for_each attribute, which accepts a map (with string keys). Set or List of strings are not supported.
- No, use of for_each or count argument is not supported within a module block.
- Yes, use the for_each or the count argument to create multiple instances of a module from a single module block similar to the resource block.
(Correct)
- Yes, this feature was added to Terraform 0.11.
- No, Since multiple module instances of the same module will have the same input variables, which is not a valid use-case for a module block but valid for a resource block where you can create multiple resources of the same type.
- Yes, this feature was added to Terraform 0.13.
(Correct)

Explanation

Multiple Instances of a Module

NOTE: Module support for the for_each and count meta-arguments was added in Terraform 0.13. Previous versions can only use these arguments with individual resources.

Use the for_each or the count argument to create multiple instances of a module from a single module block. These arguments have the same syntax and type constraints as for_each and count when used with resources.

```
# my_buckets.tf
module "bucket" {
  for_each = toset(["assets", "media"])
  source   = "./publish_bucket"
  name     = "${each.key}_bucket"
}
# publish_bucket/bucket-and-cloudfront.tf
variable "name" {} # this is the input parameter of the module
resource "aws_s3_bucket" "example" {
  # Because var.name includes each.key in the calling
  # module block, its value will be different for
  # each instance of this module.
  bucket = var.name

# ...
}

resource "aws_iam_user" "deploy_user" {
  # ...
}
```

This example defines a local child module in the ./publish_bucket subdirectory. That module has configuration to create an S3 bucket. The module wraps the bucket and all the other implementation details required to configure a bucket.

Click here to know more

Question 40: Skipped
During which phase Terraform reads from data sources?

- terraform plan
(Correct)
- terraform init
- terraform import
- terraform apply

Explanation

It is important to consider that Terraform reads from data sources during the plan phase and writes the result into the plan. Thus while working with the providers like Vault where values have associated TTL, a subsequent apply will likely fail if it is run after the intermediate token has expired due to the revocation of the secrets stored in the plan.

Click here to know more about Data Sources

Click here to know more about using vault credentials in Terraform configuration

Question 41: Skipped
Which of the following is the right way to use the min function? Choose *TWO* correct answers.

- min([12], [54], [3])
- min([12, 54, 3])
- min(12, 54, 3)
(Correct)
-

min([12, 54, 3]...)
(Correct)

Explanation

min takes one or more numbers and returns the smallest number from the set.

Examples

```
> min(12, 54, 3)
3
```

If the numbers are in a list or set value, use ... to expand the collection to individual arguments:

```
> min([12, 54, 3]...)
3
```

Click here to know more about min function

Question 42: Skipped
Terraform workspaces are technically equivalent to renaming your state file.

- False
- True
(Correct)

Explanation

Workspace Internals

Workspaces are technically equivalent to renaming your state file. They aren't any more complex than that. Terraform wraps this simple notion with a set of protections and support for remote state.

For local state, Terraform stores the workspace states in a directory called terraform.tfstate.d. This directory should be treated similarly to local-only terraform.tfstate .

Click here to know more about Terraform Workspace Internals

Question 43: Skipped
The _____ command is used to show the attributes of a single resource in the Terraform state.

- terraform show
- terraform state view
- terraform state list
- terraform state show
(Correct)
- terraform show --verbose

Explanation

The terraform state show command is used to show the attributes of a single resource in the Terraform state.

Usage

Usage: terraform state show [options] ADDRESS

The command will show the attributes of a single resource in the state file that matches the given address.

This command requires an address that points to a single resource in the state

Example: Show a Resource

The example below shows a aws_instance resource named quiz_experts:

```
$ terraform state show 'aws_instance.quiz_experts'
# aws_instance.quiz_experts:
resource "aws_instance" "quiz_experts" {
    ami                          = "ami-2757f631"
    arn                          = "arn:aws:ec2:us-east-1:434529708:instance/i-0e414f7"
    associate_public_ip_address  = true
    availability_zone            = "us-east-1c"
    cpu_core_count               = 1
    cpu_threads_per_core         = 1
    disable_api_termination      = false
    ebs_optimized                = false
    get_password_data            = false
    hibernation                  = false
    id                           = "i-0e414f7"
    instance_state               = "running"
    instance_type                = "t2.micro"
    ipv6_address_count           = 0
    ipv6_addresses               = []
    monitoring                   = false
    primary_network_interface_id = "eni-08a31fec27"
```

```
    private_dns                  = "ip-172-31-34-145.ec2.internal"
    private_ip                   = "172.31.34.145"
    public_dns                   = "ec2-3-89-144-90.compute-1.amazonaws.com"
    public_ip                    = "3.89.144.90"
    secondary_private_ips        = []
    security_groups              = [
        "default",
    ]
    source_dest_check            = true
    subnet_id                    = "subnet-01a19952e"
    tenancy                      = "default"
    volume_tags                  = {}
    vpc_security_group_ids       = [
        "sg-09efd71",
    ]
credit_specification {
        cpu_credits = "standard"
    }

metadata_options {
        http_endpoint               = "enabled"
        http_put_response_hop_limit = 1
        http_tokens                 = "optional"
    }

root_block_device {
        delete_on_termination = true
        device_name           = "/dev/sda1"
        encrypted             = false
        iops                  = 100
        volume_id             = "vol-0bc88ba0"
        volume_size           = 8
        volume_type           = "gp2"
    }
}
```

Click here to know more about state show command

Question 44: Skipped

When you run Terraform in the folder /home/quiz_experts, Terraform creates the file that contains a custom JSON format that records a mapping from the Terraform resources in your templates to the representation of those resources in the real world. Where does Terraform stores this file by default?

- /home/quiz_experts/terraform.tfstates
- /home/quiz_experts/.terraform.tfstate
- /home/quiz_experts/terraform.tfstate.d
- /home/quiz_experts/terraform.tfstate
  (Correct)

Explanation

By default, when you run Terraform in the folder /home/quiz_experts, Terraform creates the file /home/quiz_experts/terraform.tfstate. This file contains a custom JSON format that records a mapping from the Terraform resources in your templates to the representation of those resources in the real world.

For example, let's say your Terraform template contained the following:

```
resource "aws_instance" "quiz_experts_vm" {
  ami           = "ami-123"
  instance_type = "t2.micro"
}
```

After running terraform apply, the terraform.tfstate file will look something like this:

```
{
  "version": 4,
  "terraform_version": "0.12.0",
  "serial": 1,
  "lineage": "1f2087f9-4b3c-1b66-65db-8b78faafc6fb",
  "outputs": {},
  "resources": [
    {
      "mode": "managed",
      "type": "aws_instance",
      "name": "quiz_experts_vm",
      "provider": "provider.aws",
      "instances": [
        {
          "schema_version": 1,
          "attributes": {
            "ami": "ami-123",
            "availability_zone": "us-east-2c",
            "id": "i-123",
            "instance_state": "running",
            "instance_type": "t2.micro",
            "(...)": "(truncated)"
          }
        }
      ]
    }
  ]
}
```

Using this simple JSON format, Terraform knows that a resource with

type aws_instance and name quiz_experts_vm corresponds to an EC2 Instance in your AWS account with ID i-123.

Click here to know more about Terraform State Mapping

Question 45: Skipped
State is a necessary requirement for Terraform to function?

- True
(Correct)
- False

Explanation

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

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

Terraform uses this local state to create plans and make changes to your infrastructure. Prior to any operation, Terraform does a refresh to update the state with the real infrastructure.

Click here to know more about Purpose of Terraform State

Question 46: Skipped
For automation in a CI environment, Lois needs to disable the colored output of the terraform plan command; however, other terraform commands can have colored output. Is there any easier way to modify this default behavior without using -no-color option every time in the command-line?

- export TF_CLI_ARGS_plan="-no-color"
(Correct)
- export TF_IN_AUTOMATION=true
- export TF_CLI_ARGS="-no-color"
- export TF_IN_AUTOMATION="-no-color"

Explanation

TF_CLI_ARGS and TF_CLI_ARGS_name

The value of TF_CLI_ARGS will specify additional arguments to the command-line. This allows easier automation in CI environments as well as modifying the default behavior of Terraform on your own system.

These arguments are inserted directly *after* the subcommand (such as plan) and *before* any flags specified directly on the command-line. This behavior ensures that flags on the command-line take precedence over environment variables.

For example, the following command: TF_CLI_ARGS="-input=false" terraform apply -force is the equivalent to manually typing: terraform apply -input=false -force.

The flag TF_CLI_ARGS affects all Terraform commands. If you specify a named command in the form of TF_CLI_ARGS_name then it will only affect that command. As an example, to specify that only plans never refresh, you can set TF_CLI_ARGS_plan="-refresh=false".

The value of the flag is parsed as if you typed it directly to the shell. Double and single quotes are allowed to capture strings and arguments will be separated by spaces otherwise.

Click here to know more about environment variables in Terraform

Question 47: Skipped
What is the primary responsibility of providers in Terraform?

- A provider is responsible for the execution of commands or scripts on a remote machine.
- None of these.
- A provider is responsible for understanding API interactions with infrastructure providers.
(Correct)
- A provider is responsible for storing metadata about the Terraform managed infrastructure.

Explanation

Providers

Terraform is used to create, manage, and update infrastructure resources such as physical machines, VMs, network switches, containers, and more. Almost any infrastructure type can be represented as a resource in Terraform.

A provider is responsible for understanding API interactions and exposing resources. Providers generally are an IaaS (e.g. Alibaba Cloud, AWS, GCP, Microsoft Azure, OpenStack), PaaS (e.g. Heroku), or SaaS services (e.g. Terraform Cloud, DNSimple, Cloudflare).

Click here to know more about Terraform Providers

Click here to know more about Terraform Providers Types

Question 48: Skipped
Does terraform taint command modify infrastructure as soon as you run it?

- No, this command only lists all tainted resources and does not modify infrastructure or state.
- Yes, this will modify infrastructure immediately only if you are on Terraform Cloud
- No, this command *will not* modify infrastructure, but does modify the state file in order to mark a resource as tainted and the next apply will implement this change.
(Correct)
- Yes, this command will modify infrastructure by forcing labeled resources to be destroyed and recreated immediately.
- No, this is not a valid command for open-sourced Terraform command.

Explanation

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.

Click here to know more about terraform taint command

Question 49: Skipped
Misconfigured S3 buckets have resulted in large-scale data breaches in many major organizations and continue to compromise data security at a tremendous pace. Hence, Security Architect mandate all the Terraform configurations that create an S3 bucket must have encryption enabled. What is the Terraform recommended way to achieve it?

- Create a lambda function triggered on a create bucket CloudTrail event and enable encryption on the S3 bucket.
- Use Sentinel Policy.
(Correct)
- Create a best-practices document and share it with the team.
- Create a Git hook that checks the encryption parameter is enabled on an S3 bucket on every commit or pull request.

Explanation

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.

Sentinel can enforce compliance policies and policies are checked when a run is performed, after the terraform plan but before it can be confirmed or the terraform apply is executed.

Click here to know more about Sentinel

Question 50: Skipped
What programming language will you need to use to contribute to the Terraform Open Source Project?

- Java
- Bash Shell
- Go
(Correct)
- Python

Explanation

Terraform and Terraform Plugins are written in the Go programming language.

Click here to know more

Question 51: Skipped
Peter Griffin used Terraform to spin up a disposable environment for a product demo. The demo is complete now; hence he needs to destroy all the resources he created for this demo to avoid hourly charges. Since he also has few production instances on the same AWS account, he wants to review resources before deleting to be sure Terraform will not cause any unforeseen downtime. What are the options for doing that? Choose *TWO* correct answers.

- terraform destroy
  (Correct)
- terraform destroy -auto-approve
- terraform validate
- terraform plan -destroy
  (Correct)
- terraform plan

Explanation

terraform plan -destroy

Plan command with -destroy option generates a plan to destroy all the known resources.

terraform destroy

This generates a plan to destroy all the known resources and ask for confirmation to go further to destroy resources.

terraform plan

Terraform plan is not applicable here. It is used to create an execution plan to determine what actions are necessary to achieve the desired state specified in the configuration files, which is not the case here. Here we want to destroy everything mentioned in the configuration files.

terraform destroy -auto-approve

Destroy command with -auto-approve option does not generate a plan it directory destroys all resources given in the configuration files in non-intractive mode.

terraform validate

This command has no relation with resource destruction as it only 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.

Click here to know more about plan command

Click here to know more about destroy command

Click here to know more about validate command

Question 52: Skipped
When working with Terraform in a team, the use of a local file makes Terraform usage complicated. Each user must make sure they always have the latest state data before running Terraform. Which Terraform feature can overcome this complication?

- Remote State
  (Correct)
- Module
- Sentinel
- Provider

Explanation

Remote State

By default, Terraform stores state locally in a file named terraform.tfstate. When working with Terraform in a team, use of a local file makes Terraform usage complicated because each user must make sure they always have the latest state data before running Terraform and make sure that nobody else runs Terraform at the same time.

With *remote* state, Terraform writes the state data to a remote data store, which can then be shared between all members of a team. Terraform supports storing state in Terraform Cloud, HashiCorp Consul, Amazon S3, Alibaba Cloud OSS, and more.

Remote state is a feature of backends. Configuring and using remote backends is easy and you can get started with remote state quickly. If you then want to migrate back to using local state, backends make that easy as well.

Click here to know more about Remote State

Question 53: Skipped
What are all connection types supported by remote-exec provisioner to invoke a script on a remote resource after creating it? Choose *TWO* correct answers.

- winrm
  (Correct)
- telnet
- rlogin

- rdp
- rpc
- ssh
  (Correct)

Explanation

The remote-exec provisioner 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. To invoke a local process, see the local-exec provisioner instead. The remote-exec provisioner supports both ssh and winrm type connections.

```
resource "aws_instance" "quiz_experts" {
  ami           = "ami-0450ea6e7c"
  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"
    ]
  }
}
```

NOTE: Provisioners should only be used as a last resort

Click here to know more

Question 54: Skipped
When TF_LOG_PATH is set to a log file to persist logs, TF_LOG must also be set in order for any logging to be enabled.

- False
- True
  (Correct)

Explanation

Debugging Terraform

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.

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.

For example, to always write the detailed logs to the directory you're currently running terraform from:

```
export TF_LOG=TRACE
export TF_LOG_PATH=./terraform.log
```

Click here to know more

Question 55: Skipped
By default, provisioners that fail will also cause the Terraform apply itself to fail. How will you modify the resource given below to prevent Terraform from failing on the provisioner failure?

```
resource "aws_instance" "quiz_experts" {
# ...

provisioner "local-exec" {
command    = "Some command that will fail during apply"
}
}
```

-

```
resource "aws_instance" "quiz_experts" {
# ...

provisioner "local-exec" {
command    = "Some command that will fail during apply"
at_failure = continue
}
}
```

-

```
resource "aws_instance" "quiz_experts" {
# ...

provisioner "local-exec" {
```

```
command    = "Some command that will fail during apply"
on_failure = continue
}
}
```

(Correct)

```
resource "aws_instance" "quiz_experts" {
# ...

provisioner "local-exec" {
command    = "Some command that will fail during apply"
skip       = fail
}
}
```

```
resource "aws_instance" "quiz_experts" {
# ...

provisioner "local-exec" {
command    = "Some command that will fail during apply"
on_failure = fail
}
}
```

```
resource "aws_instance" "quiz_experts" {
# ...

provisioner "local-exec" {
command    = "Some command that will fail during apply"
fail_exec  = false
}
}
```

Explanation

Failure Behavior

By default, provisioners that fail will also cause the Terraform apply itself to fail. The on_failure setting can be used to change this. The allowed values are:

* continue - Ignore the error and continue with creation or destruction.

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

Example:

```
resource "aws_instance" "quiz_experts" {
 # ...
provisioner "local-exec" {
   command    = "Some command that will fail during apply"
   on_failure = continue
 }
}
```

Click here to know more about Provisioners failure behavior

Question 56: Skipped
Terraform can manage low-level components such as compute instances, storage, and networking, as well as high-level components such as DNS entries, SaaS features, etc.

False

True
(Correct)

Explanation

What is Terraform?

Terraform is a tool for building, changing, and versioning infrastructure safely and efficiently. Terraform can manage existing and popular service providers as well as custom in-house solutions.

Configuration files describe to Terraform the components needed to run a single application or your entire datacenter. Terraform generates an execution plan describing what it will do to reach the desired state, and then executes it to build the described infrastructure. As the configuration changes, Terraform is able to determine what changed and create incremental execution plans which can be applied.

The infrastructure Terraform can manage includes low-level components such as compute instances, storage, and networking, as well as high-level components such as DNS entries, SaaS features, etc.

Click here to see a video to know more about Terraform

Question 57: Skipped
You want to use an archive stored in S3 as a module source. Should the archive on s3 always have to be public to be able to be used by Terraform?

```
module "vpc" {
source = "s3::https://s3-eu-west-1.amazonaws.com/quiz-experts-s3/vpc.zip"
}
```

Yes

- No
(Correct)

Explanation

You can use archives stored in S3 as module sources using the special s3:: prefix.
The s3:: prefix causes Terraform to use AWS-style authentication when accessing the given URL.

The module installer looks for AWS credentials in the following locations, preferring those earlier in the list when multiple are available:

- The AWS_ACCESS_KEY_ID and AWS_SECRET_ACCESS_KEY environment variables.

- The default profile in the .aws/credentials file in your home directory.

- If running on an EC2 instance, temporary credentials associated with the instance's IAM Instance Profile.

NOTE: The resulting object must be an archive with one of the same file extensions as for archives over standard HTTP. Terraform will extract the archive to obtain the module source tree.

Click here to know more about Sourcing Modules from S3 / GCS Bucket

From <https://cognizant.udemy.com/course/hashicorp-certified-terraform-associate-2020/learn/quiz/4968592/result/403245958#overview>

Question 1: Skipped
Not all providers and resources support Terraform import.

- False
- True
(Correct)

Explanation

It is true, not all providers and resources support Terraform import.

Click here to know more about Terraform import limitations

Question 2: Skipped
Peter created a security group and an instance with Terraform that hosts an internal application listening on port 80. After an update, the application now listens on port 80 as well as port 443; hence, his boss has asked him to update security group rules accordingly to receive connections on both ports. Peter has changed the Terraform configuration to allow both ports, and went for a coffee break. Meantime someone updated the security group with 443 port manually in the console. Now, what will happen when Peter comes back and executes terraform apply command?

- terraform apply command will apply the same changes again.
- terraform apply command will throw an error.
- terraform apply command will do nothing.
(Correct)
- terraform apply command will add a duplicate rule.

Explanation

Before every plan or apply operation, Terraform does a refresh to update the state file with real-world status. It does this by querying infrastructure providers to find out what's actually running, and updating the state file with this new information.

Now that the state file is up to date, Terraform can compare the desired state, defined in your configuration, with the actual state of your existing resources. This comparison allows Terraform to detect which resources need to be created, modified, or destroyed.

Therefore, even after having manually updated rule on the security group using the console, Terraform will see real-world security group has the same set rules as defined in Terraform configuration; hence will do nothing during apply.

NOTE: Terraform is designed to co-exist with other tools as well as manually provisioned resources and so it only refreshes resources under its management.

Click here to know more about detecting and managing drift with Terraform

Question 3: Skipped
Which of the following terraform commands run refresh first, prior to any other work unless explicitly disabled? Choose *THREE* correct answers.

- terraform apply
(Correct)
- terraform destroy
(Correct)
- terraform plan
(Correct)
- terraform init
- terraform import

Explanation

All plan , apply and destroy commands run refresh first, prior to any other work unless explicitly disabled. Detect drift with terraform plan, which reconciles desired configuration with real-world state and tells you what Terraform will do during terraform apply in a similar way reconciled state will tell Terraform what to destroy during terraform destroy.

Terraform provides a flag to avoid refreshing the state during plan/apply/destroy, -refresh=false

Example:

```
terraform destroy -refresh=false
```

BONUS POINT: For larger infrastructures, state refresh, or querying every resource to update the state becomes too slow. Use the -refresh=false flag in order to work around this. In these scenarios, the cached state is treated as the record of truth.

Click here to know a real scenario of why you may need to use -refresh=false with destroy command

Click here to read a use case of -refresh=false

Click here to know more about it

Question 4: Skipped
By default, where does Terraform store changes after applying a configuration?

- terraform.tfstate sub-directory under the current working directory.
- terraform.tfstate.d sub-directory under the current working directory.
- terraform.tfstates file under the current working directory.
- terraform.tfstate file under the current working directory.
(Correct)

Explanation

Changes are stored by default in a local file named "terraform.tfstate" under the current working directory.

Click here to know more about Terraform state

Question 5: Skipped
The terraform refresh command is used to reconcile the state Terraform knows via its _____ with the real-world infrastructure.

- .tf files
- Provider
- State file
(Correct)
- Configuration file

Explanation

The terraform refresh command is used to reconcile the state Terraform knows about (via its **state file**) with the real-world infrastructure. This command attempts to find any resources held in the **state file** and update with any drift that has happened in the provider outside of Terraform since it was last executed.

This 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.

Click here to know more about the refresh command

Question 6: Skipped
It is mandatory to define version meta-arguments while sourcing modules from a Terraform registry.

- False

(Correct)

- True

Explanation

When using modules installed from a module registry, Terraform recommend explicitly constraining the acceptable version numbers to avoid unexpected or unwanted changes.

Hence, version meta-arguments is not mandatory, but it is highly recommended to avoid pulling breaking changes.

Use the version attribute in the module block to specify versions:

```
module "consul" {
  source  = "hashicorp/consul/aws"
  version = "0.0.5"
servers = 3
}
```

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

Click here to know more about Module versions

From <https://cognizant.udemy.com/course/hashicorp-certified-terraform-associate-2020/learn/quiz/5024206/result/403245674#overview>

Question 1: Skipped
The Terraform provisioner block works only within the resource configuration block.

- False
- True
(Correct)

Explanation

If you need to run provisioners that aren't directly associated with a specific resource, you can associate them with a null_resource.

Instances of null_resource are treated like normal resources, but they don't do anything. Like with any other resource, you can configure provisioners and connection details on a null_resource. You can also use its triggers argument and any meta-arguments to control exactly where in the dependency graph its provisioners will run.

Click here to know more

Question 2: Skipped
Let's say your company uses Terraform to create and manage resources on the AWS cloud, and your colleague needs to create two EC2 instances in two different AWS regions and creates a Terraform configuration that looks like the code given below and send it to you for review.

Looking at the code, What do you think Terraform will do? Choose *TWO* correct answers.

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

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

# 1st VM in us-east-1 region
resource "aws_instance" "quiz_experts_east" {
ami          = "some_ami_id_from_us-east-1"
instance_type = "t3.micro"

tags = {
Name = "Quiz Experts"
}
}

# 2nd VM in us-west-2 region
resource "aws_instance" "quiz_experts_west" {
ami          = "some_ami_id_from_us-west-2"
instance_type = "t3.micro"
```

```
tags = {
Name = "Quiz Experts"
}
}
```

- Terraform will create the 1st VM in "us-east-1"
  (Correct)
- Terraform will create the 2nd VM in "us-west-2"
- Terraform will create both VMs in the "us-east-1" region.
- Terraform will not throw any error on std-output.
- Terraform will try to create the 2nd VM in "us-east-1"
  (Correct)
- Terraform will work as expected and create one VM in "us-east-1" and one in "us-west-2".
- Terraform will try to create the 2nd VM in "us-west-2"
- Terraform will create both VMs in the "us-west-2" region.
- Terraform will not create any VM.

Explanation

The condition given in the question user expects to create two VMs in two different regions; hence two separate provider blocks have been defined, each for a different region.

By default, resources use a default provider configuration (one without an alias argument). Hence Terraform will try to create both VMs in "us-east-1". It will create the 1st VM as given AMI will be present in "us-east-1," but it will fail while trying to create the 2nd VM as 2nd AMI will not be visible in the "us-east-1" region.

To use an alternate provider configuration for a resource or data source, set its provider meta-argument to a <PROVIDER NAME>.<ALIAS> reference:

```
resource "aws_instance" "quiz_experts_west" {
  provider = aws.west
# ...
}
```

Hence after modification configuration given in the question will look like this:

```
provider "aws" {
  region = "us-east-1"
}
provider "aws" {
  alias  = "west"
  region = "us-west-2"
}

# 1st VM in us-east-1 region
resource "aws_instance" "quiz_experts_east" {
  ami           = "some_ami_id_from_us-east-1"
  instance_type = "t3.micro"

tags = {
    Name = "Quiz Experts"
  }
}

# 2nd VM in us-west-2 region
resource "aws_instance" "quiz_experts_west" {
  provider      = aws.west        # Add this to the 2nd VM
  ami           = "some_ami_id_from_us-west-2"
  instance_type = "t3.micro"

tags = {
    Name = "Quiz Experts"
  }
}
```

Click here to know more about Selecting Alternate Provider Configurations

Question 3: Skipped
The terraform import command can not import resources into modules but can import resources directly into your state.

- True
- False
  (Correct)

Explanation

The import command can import resources into modules as well as directly into the root of your state.

Click here to know more about it

Question 4: Skipped
Why should you constrain the acceptable provider versions in Terraform configuration for production use?

- Providers are plugins released on a separate rhythm from Terraform, so providers have their own version numbers, and version constraint ensures that provider version is the same as the Terraform version.
- Providers are plugins released on a separate rhythm from Terraform, so providers have their own version numbers, and version constraint ensures that new versions with breaking changes will not be automatically installed by terraform init in the future.
(Correct)
- Provide version constraints make sure only matching versions are cached under the plugin directory and remove other versions to optimized disk space usage.
- Provide version constraints make sure third-party provider plugins are as per requirement; however, version constraints have no significance for HashiCorp distributed providers.

Explanation

Provider Versions

Providers are plugins released on a separate rhythm from Terraform itself, and so they have their own version numbers. For production use, you should constrain the acceptable provider versions via configuration, to ensure that new versions with breaking changes will not be automatically installed by terraform init in future.

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

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

Use the terraform providers command to view the specified version constraints for all providers used in the current configuration.

Question 5: Skipped
Terraform needs GO as a pre-requisite for installation?

- False
(Correct)
- True

Explanation

You don't need to install GO for Terraform installation. Terraform is distributed as a single binary. Download and install Terraform by unzipping it and moving it to a directory included in your system's PATH.

Click here to watch video to know more about Terraform installation

Question 6: Skipped
terraform state mv command can also move resources to an entirely new state.

- True
(Correct)
- False

Explanation

terraform state mv [options] SOURCE DESTINATION

This command will move an item matched by the address given to the destination address. This command can also move to a destination address in a completely different state file.

This can be used for simple resource renaming, moving items to and from a module, moving entire modules, and more. And because this command can also move data to a completely new state, it can also be used for refactoring one configuration into multiple separately managed Terraform configurations.

This command will output a backup copy of the state prior to saving any changes. The backup cannot be disabled. Due to the destructive nature of this command, backups are required.

If you're moving an item to a different state file, a backup will be created for each state file.

Click here to know more about move command

Question 7: Skipped
Which of the following option is not right about Terraform?

- Terraform can manage existing and popular service providers as well as custom in-house solutions.
- You need knowledge of languages like Python or Go to use Infrastructure as Code tool like Terraform.
(Correct)
- Terraform is a tool for building, changing, and versioning infrastructure safely and efficiently.

- Terraform generates an execution plan describing what it will do to reach the desired state, and then executes it to build the described infrastructure. As the configuration changes, Terraform can determine what changed and create incremental execution plans that can be applied.

Explanation

The key features of Terraform are:

Infrastructure as Code

Infrastructure is described using a high-level configuration syntax. This allows a blueprint of your datacenter to be versioned and treated as you would any other code. Additionally, infrastructure can be shared and re-used.

Execution Plans

Terraform has a "planning" step where it generates an *execution plan*. The execution plan shows what Terraform will do when you call apply. This lets you avoid any surprises when Terraform manipulates infrastructure.

Resource Graph

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.

Change Automation

Complex changesets can be applied to your infrastructure with minimal human interaction. With the previously mentioned execution plan and resource graph, you know exactly what Terraform will change and in what order, avoiding many possible human errors.

Click here to know more about Terraform

Question 8: Skipped
If the Terraform backend does not have state lock enabled by default, we can use -lock flag to automatically enable state locking on all operations that could write state.

- False
(Correct)
- True

Explanation

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

If acquiring the lock is taking longer than expected, Terraform will output a status message. If Terraform doesn't output a message, state locking is still occurring if your backend supports it.

NOTE: Not all backends support locking. Please view the list of backend types for details on whether a backend supports locking or not.

Click here to know more about Terraform State

Question 9: Skipped
terraform fmt formats configuration and returns the names of the files it formatted. If configuration files are already formatted correctly, it doesn't return any file names.

- False
- True
(Correct)

Explanation

Format and validate the configuration

We recommend using consistent formatting in files and modules written by different teams. The terraform fmt command automatically updates configurations in the current directory for easy readability and consistency.

Format your configuration. Terraform will return the names of the files it formatted. If your configuration file is already formatted correctly, Terraform won't return any file names.

Click here to know more

Question 10: Skipped
How can you export Terraform debug logs to a file /var/log/terraform.log?

247. export TF_LOG="/var/log/terraform.log"

248. export TF_DEBUG_LOG="/var/log/terraform.log"

249. export TF_LOG_FILE="/var/log/terraform.log"

250. export TF_LOG_PATH="/var/log/terraform.log"
(Correct)

251. export TF_PATH="/var/log/terraform.log"

252. export TF_DEBUG_PATH="/var/log/terraform.log"

Explanation

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.

```
export TF_LOG_PATH="/var/log/terraform.log"
```

Click here to know more

Question 11: Skipped
source meta-argument is not required for a local child module.

True

False
(Correct)

Explanation

All modules require a source argument, which is a meta-argument defined by Terraform. Its value is either the path to a local directory containing 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.

Click here to know more about calling a child module

Question 12: Skipped
A module can access all the variables of the parent module.

True

False
(Correct)

Explanation

A module can not access all parent module variables; hence to pass variables to a child module, the calling module should pass specific values in the module block.

Example:

```
module "servers" {
  source = "./app-cluster"
servers = 5
}
```

A module that includes a module block like this is the *calling module* of the child module.

Click here to know more about calling a child module

Question 13: Skipped
Which of the following is the most verbose log level in Terraform?

INFO

WARN

DEBUG

TRACE
(Correct)

Explanation

Debugging Terraform

Terraform has detailed logs that 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.

Example

export TF_LOG=TRACE

Click here to know more about Terraform log levels

Question 14: Skipped
What feature of Terraform allows convenient switching between multiple instances of a single configuration within its single backend?

- Remote backend
- Providers
- Named workspaces
  (Correct)
- Sentinel

Explanation

When to use Multiple Workspaces

Named workspaces allow conveniently switching between multiple instances of a *single* configuration within its *single* backend. They are convenient in a number of situations, but cannot solve all problems.

A common use for multiple workspaces is 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. For example, a developer working on a complex set of infrastructure changes might create a new temporary workspace in order to freely experiment with changes without affecting the default workspace.

Click here to more about Terraform Workspaces

Question 15: Skipped
What should be the right type for the variable availability_zone_names in the configuration given below? Choose *THREE* correct answers.

```
253. variable "image_id" {
254. type = string
255. }
256.
257. variable "availability_zone_names" {
258. type    = _________
259. default = ["us-west-1a", "us-west-1b"]
260. }
261.
262. resource "aws_instance" "web" {
263. count           = length(var.availability_zone_names)
264. ami             = var.image_id
265. instance_type     = "t3.micro"
266. availability_zone = var.availability_zone_names[count.index]
267.
268. tags = {
269. Name = "Quiz Experts"
270. }
271. }
```

- list(any)
  (Correct)
- list
  (Correct)
- tuple
- string
- list(string)
  (Correct)
- tuple(string)
- list([string])

Explanation

List

The type list(string) means "list of strings" identified by consecutive whole numbers starting with zero.

All elements of a collection must always be of the same type.

The keyword list is a shorthand for list(any), which accepts any element type as long as every element is the same type. This is for compatibility with older configurations; for new code, we recommend using the full form.

Why Not Tuple?

Tuple is also a sequence of elements identified by consecutive whole numbers starting with zero, but each element has to has its own type and values that match the tuple type must have *exactly* the same number of elements (no more and no fewer), and the value in each position must match the specified type for that position.

Example:

tuple([string, string, string])

tuple([string, number, bool])

NOTE: The requirement is simply a list of strings that can have variable length; hence tuple is not the right choice here, but the list is.

Click here to know more about the list

Click here to know more about the tuple

Question 16: Skipped

Is it necessary to manually write a resource configuration block for the resource created outside terraform, prior to running terraform import?

- Yes, The current implementation of Terraform import does not generate configuration. A future version of Terraform will also generate configuration.
(Correct)
- No, Terraform import generates configuration automatically.

Explanation

Currently State Only

The current implementation of Terraform import can only import resources into the state. It does not generate configuration. A future version of Terraform will also generate configuration.

Because of this, prior to running terraform import it is necessary to write manually a resource configuration block for the resource, to which the imported object will be mapped.

Click here to know more about it

Question 17: Skipped

Sentinel is a policy as a code framework that's integrated into Hashicorp enterprise products. In which phase of a Terraform run Sentinel enforced user-defined policies against infrastructure?

- Between the plan and apply phases of a Terraform run
(Correct)
- After the apply phases of a Terraform run
- Before the plan phases of a Terraform run
- During the plan phases of a Terraform run

Explanation

Sentinel Policy as Code

Sentinel is a policy as a code framework that’s integrated into Hashicorp enterprise products. Sentinel allows users to define policies that are enforced against infrastructure between the plan and apply phases of a Terraform run.

Click here to know more about Sentinel

Question 18: Skipped

Which Terraform Enterprise feature allows users to create and confidentially share infrastructure modules within an organization?

- Private Module Registry
(Correct)
- Sentinel
- Private GitHub Repository
- Terraform Registry
- VCS Providers

Explanation

Private Registries

The registry at registry.terraform.io only hosts public modules, but most organizations have some modules that can't, shouldn't, or don't need to be public.

You can load private modules directly from version control and other sources, but those sources don't support version constraints or a browsable marketplace of modules, both of which are important for enabling a producers-and-consumers content model in a large organization.

If your organization is specialized enough that teams frequently use modules created by other teams, you will benefit from a private module registry.

Click here to know more about Private Registry

Question 19: Skipped
Workspaces in Terraform Cloud and Terraform CLI are the same.

- False
(Correct)
- True

Explanation

Terraform Cloud and Terraform CLI both have features called "workspaces," but they're slightly different. CLI workspaces are alternate state files in the same working directory; they're a convenience feature for using one configuration to manage multiple similar groups of resources.

In addition to the basic Terraform content, Terraform Cloud keeps some additional data for each workspace:

State versions: Each workspace retains backups of its previous state files. Although only the current state is necessary for managing resources, the state history can be useful for tracking changes over time or recovering from problems.

Run history: When Terraform Cloud manages a workspace's Terraform runs, it retains a record of all run activity, including summaries, logs, a reference to the changes that caused the run, and user comments.

Click here to know more about workspaces in Terraform cloud

Question 20: Skipped
You created a Terraform module to launch new VMs in Cloud. During your tests in a dev environment with a few VMs terraform plan and apply commands were running fine. You started using this module on production and have launched hundreds of VMs with it, and with time, you started observing slowness during terraform plan and apply commands execution. What could be the possible reason of slowness? Choose *THREE* correct answers.

- Cloud providers have API rate-limiting, and for a large infrastructure, Terraform has to make a lot of calls to querying every resource; hence API rate-limitation stops Terraform from making many concurrent API calls at once that make a plan and apply command slow.
(Correct)
- Terraform creates one VM at a time; hence, plan and apply commands were fast during test since you only created a few VMs, but on production, you may have to create many VMs at a time, which makes it slow.
- VM running Terraform CLI has no enough resources.
- 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; hence for a large infrastructure querying every resource causes slowness during the plan and apply command.
(Correct)
- By default, for every plan and apply, Terraform will sync all resources in your state with the real world to know the current state of resources to effectively determine the changes that it needs to make to reach your desired configuration. For large infrastructures, this behavior of querying every resource makes plan and apply commands execution very slow.
(Correct)

Explanation

Performance

In addition to basic resource mapping, 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.

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 *small infrastructures*, Terraform can query your providers and sync the latest attributes from all your resources. This is the default behavior of Terraform: for every plan and apply, Terraform will sync all resources in your state.

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. On top of this, cloud providers almost always have API rate limiting so Terraform can only request a certain number of resources in a period of time. Larger users of Terraform make heavy use of the -refresh=false flag as well as the -target flag in order to work around this. In these scenarios, the cached state is treated as the record of truth.

Click here to know more about Terraform Performance

Question 21: Skipped
What is the disadvantage of dynamic blocks in a Terraform configuration?

- Dynamic blocks only work with set type.
- Dynamic blocks required a dedicated provider.
- The overuse of dynamic blocks can make configuration hard to read and maintain.
  (Correct)
- It is an experimental feature and not recommended for production use.

Explanation

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.

Click here to know best practices for dynamic blocks

Question 22: Skipped
In Terraform 0.12 terraform init cannot automatically download third-party providers.

- False
- True
  (Correct)

Explanation

Third-party Plugins

Anyone can develop and distribute their own Terraform providers. These third-party providers must be manually installed, since terraform init cannot automatically download them.

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:

Operating system	User plugins directory
Windows	`%APPDATA%\terraform.d\plugins`
All other systems	`~/.terraform.d/plugins`

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.

Providers distributed by HashiCorp can also go in the user plugins directory. If a manually installed version meets the configuration's version constraints, Terraform will use it instead of downloading that provider. This is useful in airgapped environments and when testing pre-release provider builds.

Click here to know more about terraform init command

NOTE: Terraform 0.13 can do the automatic installation of third-party (community) providers

Click here to know what's new in Terraform 0.13 beta

Question 23: Skipped
terraform init downloads the source code of modules to a directory on local disk so that it can be used by other Terraform commands.

- True
  (Correct)
- False

Explanation

The source argument in a module block tells Terraform where to find the source code for the desired child module.

Terraform uses this during the module installation step of terraform init to download the source code to a directory on local disk so that it can be used by other Terraform commands.

Click here to more about Terraform Module Sources

Question 24: Skipped
A user accidentally deleted remote backend from Terraform configuration how it will impact existing resources and state file?

- Since the old state file from remote backend is not visible to current Terraform configuration; hence existing resources no longer managed by Terraform; the next terraform apply will ignore all existing resources and recreate resources again.
- Since the old state file from remote backend is not visible to the current Terraform

configuration, the next terraform apply will delete all existing resources managed by Terraform and recreate resources again.

- terraform apply command will error out. You need to reinitialize Terraform again, and it will prompt you to migrate existing remote state to the local backend so that it can still manage resources managed by the remote state.
(Correct)

- Since .terraform directory still has the old remote backend configuration, it will always point to the same remote state.

Explanation

If you delete remote backend configuration from Terraform code and run terraform apply , it will detect backend change and throw an error. You need to reinitialize Terraform again, and it will prompt you to migrate existing remote state to the local backend so that it can still manage resources managed by the remote state.

```
iac % terraform apply

Backend reinitialization required. Please run "terraform init".
Reason: Unsetting the previously set backend "s3"

The "backend" is the interface that Terraform uses to store state,
perform operations, etc. If this message is showing up, it means that the
Terraform configuration you're using is using a custom configuration for
the Terraform backend.

Changes to backend configurations require reinitialization. This allows
Terraform to setup the new configuration, copy existing state, etc. This is
only done during "terraform init". Please run that command now then try again.

If the change reason above is incorrect, please verify your configuration
hasn't changed and try again. At this point, no changes to your existing
configuration or state have been made.


Error: Initialization required. Please see the error message above.


iac % terraform init

Initializing the backend...
Terraform has detected you're unconfiguring your previously set "s3" backend.
Do you want to copy existing state to the new backend?
  Pre-existing state was found while migrating the previous "s3" backend to the
  newly configured "local" backend. No existing state was found in the newly
  configured "local" backend. Do you want to copy this state to the new "local"
  backend? Enter "yes" to copy and "no" to start with an empty state.

  Enter a value: yes


Successfully unset the backend "s3". Terraform will now operate locally.

Initializing provider plugins...

The following providers do not have any version constraints in configuration,
so the latest version was installed.

To prevent automatic upgrades to new major versions that may contain breaking
changes, it is recommended to add version = "..." constraints to the
corresponding provider blocks in configuration, with the constraint strings
suggested below.

* provider.aws: version = "~> 3.0"

Terraform has been successfully initialized!

You may now begin working with Terraform. Try running "terraform plan" to see
any changes that are required for your infrastructure. All Terraform commands
should now work.

If you ever set or change modules or backend configuration for Terraform,
rerun this command to reinitialize your working directory. If you forget, other
commands will detect it and remind you to do so if necessary.
```

Question 25: Skipped
The default workspace cannot ever be deleted?

- True
(Correct)

- False

Explanation

Terraform starts with a single workspace named "default". This workspace is special both because it is the default and also because it cannot ever be deleted. If you've never explicitly used workspaces, then you've only ever worked on the "default" workspace.

Click here to know more about Terraform Workspaces

Question 26: Skipped
How will you share state files between team members so that each team member can access the same Terraform state files to be able to use Terraform to update infrastructure?

- Terraform does not support remote state file. If you put the states on a shared location, you have to download state files manually first to use it.

- You can use either of GitHub, GitLab, Amazon S3, or HashiCorp Consul as backend.

- The most common technique for allowing multiple team members to access a common set of files is to put them in version control (e.g. Git).

- Configure Amazon S3, Azure Storage, Google Cloud Storage, HashiCorp Consul as backend, or use Terraform Enterprise which works better in a team environment.
(Correct)

Explanation

Remote State

By default, Terraform stores state locally in a file named terraform.tfstate. When working with Terraform in a team, the use of a local file makes Terraform usage complicated because each user must make sure they always have the latest state data before running Terraform and make sure that nobody else runs Terraform at the same time.

With a *remote* state, Terraform writes the state data to a remote data store, which can then be shared between all members of a team. Terraform supports storing state in Terraform Cloud, HashiCorp Consul, Amazon S3, Alibaba Cloud OSS, and more.

Remote state is a feature of backends. Configuring and using remote backends is easy and you can get started with remote state quickly. If you then want to migrate back to using local state, backends make that easy as well.

Storing state in a Version Control - Bad Idea

The most common technique for allowing multiple team members to access a common set of files is to put them in version control (e.g. Git). With Terraform state, this is a Bad Idea for the following reasons:

Locking: Most version control systems do not provide any locking form that would prevent two team members from running terraform apply on the same state file simultaneously.

Whereas remote backends like Amazon S3, or HashiCorp Consul natively support locking. To run terraform apply, Terraform will automatically acquire a lock; if someone else is already running apply, they will already have the lock, and you will have to wait. You can run aterraform apply with the -lock-timeout=<TIME> parameter to tell Terraform to wait up to TIME for a lock to be released (e.g., -lock-timeout=10m will wait for 10 minutes).

Secrets: Terraform state files are stored in plain text. This is a problem because certain Terraform resources need to store sensitive data. like username and password in the state files.

Remote backends like Amazon S3 natively support encryption in transit and encryption on disk of the state file. Moreover, those backends usually expose ways to configure access permissions (e.g., using IAM policies with an S3 bucket), so you can control who has access to your state files and the secrets they may contain. It would still be better if Terraform natively supported encrypting secrets within the state file, but these remote backends reduce most of the security concerns, as at least the state file isn't stored in plaintext on disk anywhere.

Complexity: Suppose we use a CI/CD Pipeline that deploys our application using Terraform and gets triggered by changes to the repository. Making it commit the state to the repository will make the Pipeline trigger itself again and again in an infinite deployment loop, to avoid it, we have to manually create some exceptions to CI/CD Pipeline, which adds much complexity to our automation.

Because of this, Hashicorp Terraform recommends to store state remotely instead of keeping it within a codebase. It's a much more clean and straightforward solution.

You could also use Terraform Enterprise, a Terraform paid solution that will help you out with sharing state between team members with lots of other features.

Click here to know more about Terraform Remote State

Click here to know Terraform Recommendations for Sensitive Data in State

Question 27: Skipped
Additional provider configurations are *never* inherited automatically by child modules.

- False
- True
(Correct)

Explanation

Additional provider configurations (those with the alias argument set) are *never* inherited automatically by child modules, and so must always be passed explicitly using the providers map. For example, a module that configures connectivity between networks in two AWS regions is likely to need both a source and a destination region. In that case, the root module may look something like this:

```
provider "aws" {
  alias  = "usw1"
  region = "us-west-1"
}
provider "aws" {
  alias  = "usw2"
  region = "us-west-2"
}

module "tunnel" {
  source    = "./tunnel"
  providers = {
    aws.src = aws.usw1
    aws.dst = aws.usw2
  }
}
```

The subdirectory ./tunnel must then contain *proxy configuration blocks* like the following, to declare that it requires its calling module to pass configurations with these names in its providers argument:

```
provider "aws" {
  alias = "src"
}
provider "aws" {
  alias = "dst"
}
```

Each resource should then have its own provider attribute set to either aws.src or aws.dst to choose which of the two provider configurations to use.

Click here to know more about it

Question 28: Skipped

Terraform allows a single configuration to be used to manage resources in multiple providers, and even handle cross-cloud dependencies.

- False
- True
  (Correct)

Explanation

It's often attractive to spread infrastructure across multiple clouds to increase fault-tolerance. 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.

Realizing multi-cloud deployments can be very challenging as many existing tools for infrastructure management are cloud-specific. Terraform is cloud-agnostic and allows a single configuration to be used to manage multiple providers, and to even handle cross-cloud dependencies. This simplifies management and orchestration, helping operators build large-scale multi-cloud infrastructures.

Click here to read the official Terraform document.

Question 29: Skipped

Due to some problem, you end up locking the Terraform state. Now your team members are not able to run terraform apply command to make any infrastructure changes. What will you do to fix this?

- Update lock settings in backend.tf and run terraform apply again.
- Use state-unlock command to unlock the state manually.
- Use --ignore-lock flag with terraform apply to ignore the stale lock.
- Use force-unlock command to unlock the state manually.
  (Correct)

Explanation

Force Unlock

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.

Click here to know more about force-unlock command

Question 30: Skipped

Out of all features listed below, select all the paid features. Choose *THREE* correct answers.

- Encrypted Remote State
- Remote Runs
- Roles/Team Management
  (Correct)
- Cost Estimation
  (Correct)
- Sentinel Policy as Code Management
  (Correct)
- Private Module Registry
- Secure Variable Storage

Explanation

Feature Matrix

Collaborative Infrastructure as Code	OSS	FREE	TEAM & GOVERNANCE	BUSINESS	ENTERPRISE
Remote State		✓	✓	✓	✓
VCS Connection		✓	✓	✓	✓
Workspace Management		✓	✓	✓	✓
Secure Variable Storage		✓	✓	✓	✓
Remote Runs		✓	✓	✓	✓
Private Module Registry		✓	✓	✓	✓
Team Management & Governance		FREE	TEAM & GOVERNANCE	BUSINESS	ENTERPRISE
Team Management			✓	✓	✓
Sentinel Policy as Code Management			✓	✓	✓
Cost Estimation			✓	✓	✓

Click here to see complete free and paid features list

Question 31: Skipped
Which of the following statements are correct about the Terraform state file format? Choose *FOUR* correct answers.

- The format of the state files is just YAML, which helps users modify it easily by hand whenever they want.
- Direct state file editing is risky, and manul changes can corrupt states if you make a mistake modifying your state.
(Correct)
- You should only modify state files by hand in the case of a Terraform bug.
(Correct)
- The JSON format of state files makes it easy to write tools around the state.
(Correct)
- The format of the state files is just JSON, direct file editing of the state is discouraged.
(Correct)

Explanation

The format of the state files are just JSON, direct file editing of the state is discouraged. Terraform provides the terraform state command to perform basic modifications of the state using the CLI.

The CLI manages backups for you automatically. If you make a mistake modifying your state, the state CLI will always have a backup available for you that you can restore.

The JSON format makes it easy to write tools around the state if you want or to modify it by hand in the case of a Terraform bug.

Click here to know more about Terraform State Format

Question 32: Skipped
What are the features exclusive to Terraform Enterprise. Select all that apply. Choose *TWO* correct answers.

- Audit logging
(Correct)
- Cost Estimation
- SAML single sign-on
(Correct)
- Sentinel

Explanation

Terraform Enterprise is our self-hosted distribution of Terraform Cloud. It offers enterprises a private instance of the Terraform Cloud application, with no resource limits and with additional enterprise-grade architectural features like audit logging and SAML single sign-on.

Click here to know more about Terraform Enterprise

Question 33: Skipped
Which of the following is right about Terraform? Choose *THREE* correct answers.

- Terraform can help team creating disposable parallel environments.
(Correct)
- Terraform can codified infrastructure configuration; however, configuration sharing within an organization still not possible.
- Customer teams can use shared Terraform configurations and use them as a black box tool to manage their services.
(Correct)
- Terraform is not limited to physical providers like AWS. Resource schedulers can be treated as a provider, enabling Terraform to request resources from them.
(Correct)
-

Terraform is not cloud-agnostic

Explanation

Disposable Environments

It is common practice to have both a production and staging or QA environment. These environments are smaller clones of their production counterpart, but are used to test new applications before releasing in production. Using Terraform, the production environment can be codified and then shared with staging, QA or dev. These configurations can be used to rapidly spin up new environments to test in, and then be easily disposed of. Terraform can help tame the difficulty of maintaining parallel environments, and makes it practical to elastically create and destroy them.

Resource Schedulers

In large-scale infrastructures, static assignment of applications to machines becomes increasingly challenging. To solve that problem, there are a number of schedulers like Borg, Mesos, YARN, and Kubernetes. These can be used to dynamically schedule Docker containers, Hadoop, Spark, and many other software tools.

Terraform is not limited to physical providers like AWS. Resource schedulers can be treated as a provider, enabling Terraform to request resources from them. This allows Terraform to be used in layers: to set up the physical infrastructure running the schedulers as well as provisioning onto the scheduled grid.

Multi-Cloud Deployment

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

Self-Service Clusters

At a certain organizational size, it becomes very challenging for a centralized operations team to manage a large and growing infrastructure. Instead, it becomes more attractive to make "self-serve" infrastructure, allowing product teams to manage their own infrastructure using tooling provided by the central operations team.

Using Terraform, the knowledge of how to build and scale a service can be codified in a configuration. Terraform configurations can be shared within an organization enabling customer teams to use the configuration as a black box and use Terraform as a tool to manage their services.

Click here to know more Terraform use cases

Question 34: Skipped

Your staging and production environment points to the same module, which is making it harder to test a change in staging without any chance of affecting production. Your colleague suggested you to use *versioned modules* so that you can use one version in staging and a different version in production. How can you use this approach if you are sourcing modules from a private git repo?

- Create separate tags and use version meta-argument to specify different tags for production and staging environment.
- You can only source modules from a default branch; hence you must use a separate repo for each environment.
- None of these
- Modules sourced from git does not support it. You must use versioning enabled S3.
- Create separate tags and use a stable one for the production and a new one for the staging.
  (Correct)
- This is a paid feature only supported by Terraform Cloud and Terraform Enterprise

Explanation

By default, Terraform will clone and use the default branch (referenced by HEAD) in the selected repository. You can override this using the ref argument.

The ref parameter allows you to specify a particular Git commit sha1 hash, a branch name, or a specific Git tag. A tag is just a pointer to a commit, but it allows you to use a friendly, human-readable name.

For example, if you need to use Git tag v1.2.0, you can modify the source URL under the module block, as given below.

```
module "vpc" {
  source = "git::https://code.quizexperts.com/vpc.git?ref=v1.2.0"
}
```

NOTE: The value of the ref argument can be any reference that would be accepted by the git checkout command, including branch and tag names.

Click here to know more about Selecting a Revision

NOTE: This is an important topic for the exam.

Question 35: Skipped
A module must be public to be able to be used in your terraform configuration.

- True
- False
  (Correct)

Explanation

Module Sources

The source argument in a module block tells Terraform where to find the source code for the desired child module.

A module can be public or private.

The module installer supports installation from a number of different source types, as listed below.

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

Question 36: Skipped
To ensure that Terraform workspace names are stored correctly and safely in all backends, the workspace name must be valid to use in a URL path segment without escaping.

- True
  (Correct)
- False

Explanation

Workspace Internals

Workspaces are technically equivalent to renaming your state file. They aren't any more complex than that. Terraform wraps this simple notion with a set of protections and support for remote state.

For local state, Terraform stores the workspace states in a directory called terraform.tfstate.d. This directory should be treated similarly to local-only terraform.tfstate; some teams commit these files to version control, although using a remote backend instead is recommended when there are multiple collaborators.

For remote state, the workspaces are stored directly in the configured backend. For example, if you use Consul, the workspaces are stored by appending the workspace name to the state path. To ensure that workspace names are stored correctly and safely in all backends, the name must be valid to use in a URL path segment without escaping.

The important thing about workspace internals is that workspaces are meant to be a shared resource. They aren't a private, local-only notion (unless you're using purely local state and not committing it).

The "current workspace" name is stored only locally in the ignored .terraform directory. This allows multiple team members to work on different workspaces concurrently. The "current workspace" name is **not** currently meaningful in Terraform Cloud workspaces since it will always have the value default.

Click here to read the official document

Question 37: Skipped
How to verify configuration in the context of a particular run and validate if all variables are defined and not missing input values? Choose *TWO* correct answers.

- terraform init
- terraform fmt
- terraform validate
  (Correct)
- terraform plan
  (Correct)

Explanation

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 the correctness of attribute names and value types.

To verify configuration in the context of a particular run (a particular target workspace, input variable values, etc), use the terraform plan command instead, which includes an implied validation check.

Click here to know more about Configuration Validator

Question 38: Skipped
Which of the following configuration block has Terraform 0.13 recommended way of provider version constraint? Choose *THREE* correct answers.

```
# Provider information Option 1

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

```
# Provider information Option 2

terraform {
required_providers {
aws = {
source  = "hashicorp/aws"
version = ">= 3.0"
}
}
}
```

- The use of *string* version constraint within required_providers block is recommended in Terraform 0.13 or later.
- The use of an *object* version constraint within required_providers block is recommended in Terraform 0.13 or later.
  (Correct)
- The configuration given below is available only for backward compatibility to support the transition to Terraform 0.13.

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

  (Correct)
- The use of explicit source addresses for all providers is recommended in Terraform 0.13 or later.
  (Correct)
- The configuration given below is available only for backward compatibility to support the transition to Terraform 0.13.

```
terraform {
required_providers {
aws = {
source  = "hashicorp/aws"
version = ">= 3.0"
}
}
}
```

Explanation

A provider requirement consists of a local name, a source location, and a version constraint is recommended i Terraform 0.13 or later:

```
terraform {
  required_providers {
    aws = {
      source  = "hashicorp/aws"
      version = ">= 3.0"
    }
  }
}
```

source - the global source address for the provider you intend to use, such as hashicorp/aws.

version - a version constraint specifying which subset of available provider versions the module is compatible with.

NOTE:

- The name = { source, version } syntax for required_providers was added in Terraform v0.13. Previous versions of Terraform used a version constraint string instead of an object (like mycloud = "~> 1.0"), and had no way to specify provider source addresses.

- Prior to Terraform 0.13, providers only had local names, since Terraform could only

automatically download providers distributed by HashiCorp.

- If you omit the source argument when requiring a provider, Terraform uses an implied source address of registry.terraform.io/hashicorp/<LOCAL NAME>. This is a backward compatibility feature to support the transition to Terraform 0.13; in modules that require 0.13 or later, we recommend using explicit source addresses for all providers.

Click here to know more about Provider Requirements

Question 39: Skipped

Peter created an EC2 with the configuration given below. DevOps team of his company noticed an unencrypted volume is attached to this EC2 and encrypts it. What will happen if Peter runs terraform apply again on the same configuration?

```
#.....

resource "aws_instance" "example" {
ami          = "ami-2757f631"
instance_type = "t2.micro"
root_block_device {
volume_size          = 20
delete_on_termination = true
}
}
```

- Since encryption is still not the part desired state, defined in the configuration, hence during the next terraform apply Terraform will just update the state file and not touch the running EC2.
  (Correct)
- The attached volume is different from the Terraform state volume; hence, Terraform will recreate EC2 during the next terraform apply.
- Since state data does not match the real-world resource, there is no way to correlate this information; hence next terraform apply will result in a new duplicate EC2 and an unmanaged EC2.
- terraform apply will revert the manual changes.

Explanation

Terraform reconciles the resources tracked by the state file with the real world. It does this by querying your infrastructure providers to find out what's actually running and the current configuration, and updating the state file with this new information. Terraform is designed to co-exist with other tools and manually provisioned resources, so it only refreshes resources under its management.

The output for a refresh is minimal. Terraform lists each resource it is refreshing along with its internal ID. Running refresh does not modify infrastructure, but does modify the state file. If the state has drifted from the last time Terraform ran, refresh allows that drift to be detected.

A Terraform plan is a description of everything Terraform will do to implement your desired configuration when you apply the plan. Terraform plan is done automatically during an apply but can also be done explicitly.

In the last example, where if we updated the AMI or instance_type of the EC2, the terraform plan will indicate that the existing instance would be destroyed and recreated. However, volume encryption will result in an in-place update, which will only update the state.

Click here to know more about it

Question 40: Skipped

Modules sourced from local file paths support version meta-argument within module blocks.

- False
  (Correct)
- True

Explanation

Local Modules

Modules sourced from local file paths do not support version; since they're loaded from the same source repository, they always share the same version as their caller.

Example:

```
module "consul" {
  source = "./consul"
}
```

Terraform Registry Modules

Version constraints are supported only for modules installed from a module registry, such as the public Terraform Registry or Terraform Cloud's private module registry.

Example:

```
# Modules on the public Terraform Registry
module "consul" {
```

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

# Modules hosted in other registries

module "consul" {
  source = "app.terraform.io/quizexperts/k8s-cluster/azurerm"
  version = "1.1.0"
}
```

Other module sources

Other module sources can provide their own versioning mechanisms within the source string itself, or might not support versions at all.

Example:

```
module "vpc" {
  source = "git::https://code.quizexperts.com/vpc.git?ref=v1.2.0"
}
```

Click here to know more about Module Versions

Click here to know more about Modules Sources

Question 41: Skipped
As given below, Glenn has created a variable of *boolean* type. Select all accepted input from the options given below.

```
variable "set_password" {
type = bool
}
```

- false
- Both of these
  (Correct)
- "false"
- None of these

Explanation

The Terraform language will automatically convert string values to bool values when needed, and vice-versa as long as the string contains a valid representation of a boolean value.

* "true" converts to true, and vice-versa
* "false" converts to false, and vice-versa

The True and False keywords are not valid boolean values in Terraform language.

Click here to know more about type conversion in Terraform

Question 42: Skipped
What is the terraform state command used for? Choose *THREE* correct answers.

- Checking backend utilization state.
- Modifying/Updating Terraform state.
  (Correct)
- Moving items in/out a Terraform state.
  (Correct)
- Initializing new changes in remote state configuration.
- Advanced state management.
  (Correct)
- Importing manually created resources to Terraform state.

Explanation

State Command

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.

Click here to know more about state command

Question 43: Skipped
If you remove resources from the Terraform state, resources will be physically destroyed from real-world infrastructure.

- True
- False
(Correct)

Explanation

Removing a Resource from State

Items removed from the Terraform state are *not physically destroyed*. Items removed from the Terraform state are only no longer managed by Terraform. For example, if you remove an AWS instance from the state, the AWS instance will continue running, but terraform plan will no longer see that instance.

There are various use cases for removing items from a Terraform state file. The most common is refactoring a configuration to no longer manage that resource (perhaps moving it to another Terraform configuration/state).

The state will only be saved on successful removal of all addresses. If any specific address errors for any reason (such as a syntax error), the state will not be modified at all.

This command will output a backup copy of the state prior to saving any changes. The backup cannot be disabled. Due to the destructive nature of this command, backups are required.

Command: state rm

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.

Example: Remove a Resource

The example below removes the aws_instance resource named vm:

```
$ terraform state rm 'aws_instance.vm'
```

Click here to know more about state rm command

Question 44: Skipped
Anyone can publish and share modules on the Public Terraform Registry.

- False
- True
(Correct)

Explanation

Publishing Modules

Anyone can publish and share modules on the Terraform Registry.

Published modules support versioning, automatically generate documentation, allow browsing version histories, show examples and READMEs, and more. We recommend publishing reusable modules to a registry.

Public modules are managed via Git and GitHub. Publishing a module takes only a few minutes. Once a module is published, you can release a new version of a module by simply pushing a properly formed Git tag.

The registry extracts information about the module from the module's source. The module name, provider, documentation, inputs/outputs, and dependencies are all parsed and available via the UI or API, as well as the same information for any submodules or examples in the module's source repository.

Click here to know more about Publishing Modules

Question 45: Skipped
Refer to the configuration block given below and suggest the value of X to get ami-b374d5a5 as selected_ami *output.*

```
variable "ami_map" {
type = map
default = {
"us-east-1" = "ami-b374d5a5"
"us-west-2" = "ami-fc0b939c"
}
}

output "selected_ami" {
value = X
}
```

- var.ami_map("us-east-1")
- var.ami_map[0]["us-east-1"]
- var.ami_map[1]["us-east-1"]
- var.ami_map["us-east-1"]
(Correct)
- var.ami_map[0]

Explanation

A variable can be explicitly declared as a map type, or it can be implicitly created by specifying a default value that is a map. The above demonstrates both an explicit type = "map" and an implicit default = {}.

To use the ami_map map, edit selected_ami to use var.ami_map["us-east-1"] to produce ami-b374d5a5 as output.

The square-bracket index notation used here as an example of how the map type expression is accessed as a variable.

For a static value lookup, the region could be hard-coded such as var.ami_map["us-east-1"] .

For a dynamic value lookup with var.ami_map we can declare another variable which will hold region name so that we can get AMI value automatically based on the region variable.referencing the var.region declaration for dynamic lookup.

```
variable "ami_map" {
  type = map
  default = {
    "us-east-1" = "ami-b374d5a5"
    "us-west-2" = "ami-fc0b939c"
  }
}
output "selected_ami" {
  value = var.ami_map["us-east-1"]
}
```

For a static value lookup, the region could be hard-coded such as var.ami_map["us-east-1"] .

For a dynamic value lookup with var.ami_map , we can declare another variable that will hold a region name so that we can get AMI value automatically based on the region variable value.

```
variable "region" {}
variable "ami_map" {
  type = map
  default = {
    "us-east-1" = "ami-b374d5a5"
    "us-west-2" = "ami-fc0b939c"
  }
}
output "selected_ami" {
  value = var.ami_map[var.region]
}
```

Click here to know more about Maps

Question 46: Skipped
Which command can be used to inspect a plan to ensure that the planned operations are expected?

- terraform plan
- terraform state
- terraform inspect
- terraform show
  (Correct)

Explanation

The terraform show command is used to provide human-readable output from a state or plan file. This can be used to inspect a plan to ensure that the planned operations are expected, or to inspect the current state as Terraform sees it.

Click here to know more about show command

Question 47: Skipped
How does Terraform create provider dependencies and understand why a particular provider is needed? Choose *THREE* correct answers.

- Resource belonging to a particular provider removed from configuration but exists in the current *state*.
  (Correct)
- Terraform downloads provider plugins of all major clouds during terraform installations and save them in the user home directory.
- Explicit use of a *provider* block in configuration, optionally including a version constraint.
  (Correct)
- Use of any resource belonging to a particular *provider* in a resource or data block in the configuration.
  (Correct)
- Provider code for all major clouds is built-in in Terraform code. Terraform only download

new provider plugins if a resource belonging to a community provider is being used.

Explanation

Provider dependencies are created in several different ways:

* Explicit use of a provider block in configuration, optionally including a version constraint.

* Use of any resource belonging to a particular provider in a resource or data block in configuration.

* Existence of any resource instance belonging to a particular provider in the current *state*. For example, if a particular resource is removed from the configuration, it continues to create a dependency on its provider until its instances have been destroyed.

terraform providers command gives an overview of all of the current dependencies, as an aid to understanding why a particular provider is needed.

Click here to know more about Provider dependency

Question 48: Skipped
Which of the following can we use to make Terraform configuration dynamic and reusable?

- Modules
- Input Variables
  (Correct)
- Functions
- Data Sources

Explanation

Input variables serve as parameters for a Terraform module, allowing aspects of the module to be customized dynamically without altering the module's own source code, and allowing modules to be shared between different configurations.

Click here to know more about Variables

Question 49: Skipped
Destroy provisioner runs only during a destroy operation. Hence destroy-time provisioner within a resource will run even if the resource is tainted manually.

- False
  (Correct)
- True

Explanation

A destroy-time provisioner within a resource that is tainted *will not* run. This includes resources that are marked tainted from a failed creation-time provisioner or tainted manually using terraform taint.

Click here to know more about Destroy-Time Provisioners

Question 50: Skipped
Stewie Griffin has an application running on EC2 instance that expects to use a specific Amazon S3 bucket. This dependency is configured inside the application, and thus not visible to Terraform.

How can he explicitly declare this dependency and forcing Terraform to create S3 bucket before EC2 instance?

- Ask him to create two separate configurations for each and run the S3 bucket configuration before creating an EC2 instance.
- Ask him to use the terraform link command to declare this dependency explicitly.
- Terraform is a procedural configuration language. Hence tell him to declare resources in the order he wants Terraform to creates or destroys them.
- He can use depends_on to declare the dependency explicitly. Terraform will wait until the S3 bucket has been created before creating the EC2 instance.
  (Correct)

Explanation

He can use depends_on to declare the dependency explicitly. He can also specify multiple resources in the depends_on argument, and Terraform will wait until all of them have been created before creating the target resource.

Example:

```
resource "aws_s3_bucket" "quiz_experts_s3" {
  bucket = "quiz_experts_questions_s3"
  acl    = "private"
}
resource "aws_instance" "quiz_experts_ec2" {
  ami           = "ami-2768hd97c"
  instance_type = "t2.micro"
```

```
depends_on = [aws_s3_bucket.quiz_experts_s3]
}
```

NOTE: Since Terraform will wait to create the dependent resource until after the specified resource is created, adding explicit dependencies can increase the length of time it takes for Terraform to create infrastructure.

NOTE: The order in which resources are declared in a configuration files has no effect on the order in which Terraform creates or destroys them.

One more example of even more complicated dependency.

```
resource "aws_s3_bucket" "quiz_experts_s3" {
  bucket = "quiz_experts_questions_s3"
  acl    = "private"
}
resource "aws_instance" "quiz_experts_ec2" {
  ami           = "ami-2768hd97c"
  instance_type = "t2.micro"

depends_on = [aws_s3_bucket.quiz_experts_s3]
}

module "example_sqs_queue" {
  source  = "terraform-aws-modules/sqs/aws"
  version = "2.1.0"

depends_on = [aws_s3_bucket.quiz_experts_s3, aws_instance.quiz_experts_ec2]
}
```

Click here to know more about Terraform dependencies

Question 51: Skipped
What will be the value of output lucky_user?

```
variable "users" {
type    = list(string)
default = ["Peter", "Lois", "Brian", "Meg"]
}

output "lucky_user" {
value = element(var.users, 4)
}
```

- Peter
  (Correct)
- None of these.
- This configuration will result in an index out of range error.
- Meg

Explanation

element retrieves a single element from a list. (zero-based index)

NOTE: If the given index is greater than the length of the list then the index is "wrapped around" by taking the index modulo the length of the list:

Output of Terraform configuration given in the question:

```
% terraform apply
Apply complete! Resources: 0 added, 0 changed, 0 destroyed.

Outputs:

lucky_user = Peter
```

More Examples:

```
> element(["a", "b", "c"], 1)
b

> element(["a", "b", "c"], 3)
a
```

Click here to know more about element function

Question 52: Skipped
To create infrastructure resources given in the image provided below, you created a Terraform configuration example.tf, which looks like this -

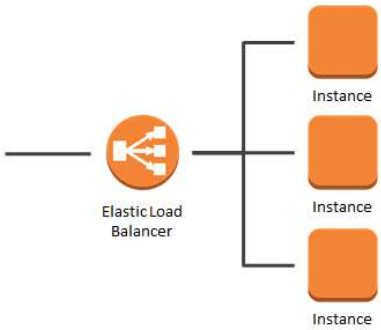
```
module "instances" {
source = "./app-cluster"

instances = 3
}

resource "aws_elb" "quiz_experts_elb" {
# ...

instances = module.instances.instance_ids
}
```

While running the terraform plan command on it, you encountered the error given below -

```
% terraform plan

Error: Reference to undeclared output value

on example.tf line 10, in resource "aws_elb" "quiz_experts_elb":
10:    instances = module.instances.instance_ids

An output value with the name "instance_ids" has not been declared in
module.instances.
```

How can you solve it?

Larger image



AWS

- Declare a variable in the instances module to export a list of instance ids as variable named instance_ids.
- Execute module code first in a different directory and once instances are up run terraform apply to attach instances to elb.
- Declare output values in the calling module to import a list of instance ids from the child module.
- Declare output values in the instances module to export a list of instance ids as output named instance_ids.
  (Correct)

Explanation

The resources defined in a module are encapsulated, so the calling module cannot access their attributes directly. However, the child module can declare output values to selectively export certain values to be accessed by the calling module.

For example, if the ./app-cluster module referenced in the example above has to export an output value named instance_ids only then the calling module can reference that result using the expression module.instances.instance_ids

```
output "instance_ids" {
  value = aws_instance.my_vm.*.id
}
```

Click here to know more about Accessing Module Output Values

Question 53: Skipped
If creation-time provisioner fails, Terraform marks the resource as _____.

- undesirable
- broken
- tainted
  (Correct)
- unprovisioned
- failed

Explanation

If a creation-time provisioner fails, the resource is marked as **tainted**. A tainted resource will be planned for destruction and recreation upon the next terraform apply. Terraform does this because a failed provisioner can leave a resource in a semi-configured state. Because Terraform cannot reason about what the provisioner does, the only way to ensure proper creation of a resource is to recreate it. This is tainting.

Click here to know more about Creation-Time Provisioner

Question 54: Skipped
Which of the following command upgrades a Terraform provider to the latest acceptable version?

- terraform providers -upgrade=true
- terraform init -update
- terraform providers -update
- terraform init -upgrade
(Correct)

Explanation

To upgrade to the latest acceptable version of each provider, run terraform init -upgrade. This command also upgrades to the latest versions of all Terraform modules.

Click here to know more about Provider Versions

Question 55: Skipped
The Terraform language supports user-defined functions.

- False
(Correct)
- True

Explanation

The Terraform language includes a number of built-in functions that you can call from within expressions to transform and combine values. The general syntax for function calls is a function name followed by comma-separated arguments in parentheses:

```
<FUNCTION NAME>(<ARGUMENT 1>, <ARGUMENT 2>)
```

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

You can experiment with the behavior of Terraform's built-in functions from the Terraform expression console, by running the terraform console command:

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

Click here to know more about functions

Question 56: Skipped
Terraform has multiple mechanisms for setting variables. If the same variable is assigned multiple values, which value Terraform will use?

- Last
(Correct)
- First
- None, Terraform will throw error If the same variable is assigned multiple values.

Explanation

Variable Definition Precedence

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.)

Click here to know more about variable definition precedence

Question 57: Skipped
In Terraform 0.13 terraform init cannot automatically download third-party providers.

- True
- False
(Correct)

Explanation

Before Terraform 0.13, Terraform could only install providers that were distributed by HashiCorp. Now Terraform 0.13 can install third-Party Providers. Hence with Terraform 0.13, terraform init will automatically download and install partner and community providers in the HashiCorp Terraform Registry, following the same clear workflow as HashiCorp-supported official providers. These improvements to the ecosystem will benefit

Terraform users and provider developers alike.

Click here to know more about Automatic Installation of Third-Party Providers with Terraform 0.13

Continue
Retake test
Fullscreen
Expand
Course content
Practice tests

- Progress cannot be changed for this item
  Practice Test 1: Terraform Associate Certification
- Progress cannot be changed for this item
  Practice Test 2: Terraform Associate Certification
- Progress cannot be changed for this item
  Practice Test 3: Terraform Associate Certification
- Progress cannot be changed for this item
  Practice Test 4: Important Questions

Overview
Q&AQuestions and answers
Notes
Announcements

From <https://cognizant.udemy.com/course/hashicorp-certified-terraform-associate-2020/learn/quiz/4969060/result/403244966#overview>

Question 1: Skipped
Why does the traditional approach of managing infrastructure not go well with large-scale, distributed systems architectures? Choose *THREE* correct answers.

- The traditional approach was okay if we didn't have to manage a lot of infrastructure. But it does not fit well where The scale of infrastructure is much larger, because instead of a handful of large instances, we might have many smaller instances, so there are many more things we need to provision.
  (Correct)
- Pointing and clicking into an administrative console is a more scalable solution for large-scale, distributed systems architectures
- The traditional approach can’t keep up with higher elasticity of infrastructure, where resources might live days to weeks instead of months to years.
  (Correct)
- Infrastructure changes are now relatively minimal, whereas the traditional approach helps managing infrastructures where changes are more often.
- Logging it into a management portal or an administrative console, and pointing and clicking to provision large infrastructure is slow and prone to human error.
  (Correct)

Explanation

Let us take an example to understand it. Your organization runs VMware, inside a private data center, and you need a new VMs. The **traditional approach** was: You would file a ticket, and then someone on the other end of this ticketing queue is pulling it off, logging it into a management portal or an administrative console, and pointing and clicking to provision that VM.

This was okay if you didn't have to manage a lot of infrastructure.

Now, as we're making a transition to a **cloud environment**, which is predominantly API-driven, the scale of infrastructure is much larger and more elastic. Instead of a handful of large instances, we might have many smaller instances, so there are many more things we need to provision, and this infrastructure tends to be cyclic. We might scale up to handle a load during a peak day, and then scale down at night to save on cost, because it's not a fixed cost. Unlike owning hardware, we're now paying by the hour. So, it makes sense only to use the infrastructure you need. You have to have this elasticity if you were using the traditional approach to handle the elastic nature of infrastructure. You have to file a thousand tickets every morning to spin up to peak capacity, then file another thousand tickets at night to spin back down, and then manually manage all of this. It slows down the process, and pointing and clicking are prone to human error.

Click here to see a video to know more

Question 2: Skipped
You create a VM with the configuration given below to host an internal tool and push this Terraform configuration to GitHub so that your teammate also can manage the same VM. Your teammate cloned this repository and saw that you named VM as testing and changed the resource name to tool_instance to make it more relatable

with the tool it hosts. Now, how will Terraform behave with VM during the next apply?

***Before*:**

```
terraform {
backend "s3" {
bucket         = "quiz-experts"
key            = "terraform/dev/states"
region         = "us-east-1"
dynamodb_table = "terraform-locks"
encrypt        = true
}
}

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

resource "aws_instance" "test" {
ami           = "ami-05794d4ef61b053f2"
instance_type = "t2.micro"
}
```

After:

```
terraform {
backend "s3" {
bucket         = "quiz-experts"
key            = "terraform/dev/states"
region         = "us-east-1"
dynamodb_table = "terraform-locks"
encrypt        = true
}
}

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

resource "aws_instance" "tool_instance" {
ami           = "ami-05794d4ef61b053f2"
instance_type = "t2.micro"
}
```

- Terraform apply after renaming a resource does not modify infrastructure, but does modify the state file.
- Terraform apply after renaming a resource will create one more resource with the new name, and the user has to delete the old resource manually.
- Even if the user has just changed the resource name, Terraform will delete the existing VM and create a new VM with a new name.
  (Correct)
- Terraform will do nothing since VM already exists in the state and in the desired state.

Explanation

Rename a Resource

Terraform tracks resources by their name. If you change the name, you have *created a new resource* and *deleted the old resource*. There is no way for Terraform to tell you changed the name of an existing resource since its all just text. Hence even if the user has just changed the resource name, Terraform will delete the existing VM and create a new VM with a new name.

How to rename a resource instead of deleting it?

Renaming a resource in Terraform without deleting and creating it is a 2 step process.

For example, to rename the `aws_instance` resource named `test` to `tool_instance`:

1) Define your resource block with a new name.

```
# Something
resource "aws_instance" "tool_instance" {
  ami           = "ami-05794d4ef61b053f2"
  instance_type = "t2.micro"
}
```

2) Use the Terraform move command.

```
# Execute the Terraform move command
# by passing your old resource name first, followed by your new resource name.
$ terraform state mv 'aws_instance.test' 'aws_instance.tool_instance'
```

To verify it, run, Terraform plan command after above given two steps, and make sure `terraform plan` says - *"No changes. Infrastructure is up-to-date."* same as example given below -

```
$ terraform plan
Acquiring state lock. This may take a few moments...
Refreshing Terraform state in-memory prior to plan...
The refreshed state will be used to calculate this plan, but will not be
persisted to local or remote state storage.
```

```
aws_instance.tool_instance: Refreshing state... [id=i-018d635d0ab1a2f4b]

------------------------------------------------------------------------

No changes. Infrastructure is up-to-date.

This means that Terraform did not detect any differences between your
configuration and real physical resources that exist. As a result, no
actions need to be performed.
```

Click here to see more resource renaming examples

Question 3: Skipped

In regards to Terraform List, Select all correct statements. Choose *THREE* correct answers.

- List is a collection of multiple values of several distinct types grouped together as a single value.
- List is a collection of values identified by a string label.
- List is a sequence of values identified by consecutive whole numbers starting with zero.
  (Correct)
- ["a", [], "b"] is a valid Terraform list.
- ["a", 15, true] is a valid Terraform list.
  (Correct)
- List is a collection of unique values.
- All elements of a list must always be of the same type.
  (Correct)

Explanation

List (**Collection Type**)

A list is a sequence of values of the same type identified by consecutive whole numbers starting with zero.

For example, the type list(string) means "list of strings", which is a different type than list(number), a list of numbers. All elements of a collection must always be of the same type.

Whenever possible, Terraform converts values between similar kinds of complex types (list/tuple/set and map/object) if the provided value is not the exact type requested, therefore -

* A tuple can be converted to a list, and vice-versa.
* An object can be converted to a map and vice-versa.

However, certain conditions should meet to enable this conversion. For examples -

Example 1:

A list can only be converted to a tuple if it has *exactly* the required number of elements.

Example 2:

If a module argument requires a value of type list(string) and a user provides the tuple ["a", 15, true], Terraform will internally transform the value to ["a", "15", "true"] by converting the elements to the required string element type as all of the elements of a collection must have the same type.

On the other hand, an automatic conversion will fail if the provided value (including any of its element values) is incompatible with the required type. If an argument requires a type of list(string) and a user provides the tuple ["a", [], "b"] then the value cannot conform to the type constraint, Terraform would reject this value, complaining that all elements must have the same type, since an empty tuple cannot be converted to a string.

NOTE: The keyword list is a shorthand for list(any), which accepts any element type as long as every element is the same type. This is for compatibility with older configurations; for new code, we recommend using the full form.

Click here to know more about Terraform list

Click here to know more about Conversion of Complex-types in Terraform

Question 4: Skipped

Your team uses Terraform to manage dev, staging, and production environments. For every update, you have to update each environment containing duplicate blocks of configuration separately. What Terraform features can you use to avoid this growing burden of configuration updates?

- Packages

- Providers
- Plugins
- Modules
  (Correct)
- Functions

Explanation

Terraform Module

As you manage your infrastructure with Terraform, you will create increasingly complex configurations. There is no intrinsic limit to the complexity of a single Terraform configuration file or directory. It is possible to continue writing and updating your configuration files in a single directory. However, if you do, you end up with a very complicated configuration with lots of duplicate blocks. Updating the configuration will become riskier, as an update to one section may cause unintended consequences to other parts of your configuration.

Terraform recommend that every practitioner use modules by following these best practices:

- 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.

- 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 significantly reduce the burden of maintaining and updating your configuration as your infrastructure grows in complexity.

- 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.

- Publish and share modules with your team. Most infrastructure is managed by a team of people, and modules are an important way for teams to work together to create and maintain infrastructure. As mentioned earlier, you can publish modules either publicly or privately.

Click here to more about Terraform Modules

Click here to know more about Module Best Practices

Question 5: Skipped
Peter is writing a module that creates VPC, subnets, security-groups, route-tables, and nat-gateway. Each of these resources has common team-specific tags. Therefore, instead of defining the same tags multiple times in the configuration. What approach should Peter use to make the configuration more readable and easy to update in the future?

- Use Local Values
  (Correct)
- Use Input Variables
- Use Functions
- Use Dynamic Blocks
- Use Data Sources

Explanation

When To Use Local Values

Local values can be helpful to avoid repeating the same values or expressions multiple times in a configuration, but if overused they can also make a configuration hard to read by future maintainers by hiding the actual values used.

Use local values only in moderation, in situations where a single value or result is used in many places *and* that value is likely to be changed in the future. The ability to easily change the value in a central place is the key advantage of local values.

Click here to know more about Local Values

Question 6: Skipped
Which of the following represents a valid source string to use a module from a private registry?

```
module "vpc" {
source = "_____________________________________"
version = "0.9.3"
}
```

- <HOSTNAME>/<NAMESPACE>/<NAME>/<PROVIDER>
  (Correct)
- <NAMESPACE>/<PROVIDER>/<NAME>
- <NAMESPACE>/<NAME>/<PROVIDER>

- <HOSTNAME>/<NAME>/<PROVIDER>
- <HOSTNAME>/<NAMESPACE>/<PROVIDER>/<NAME>

Explanation

Private Registry Module Sources

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.

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

Click here to know more about Private Registry Module Sources

Question 7: Skipped
Which command automatically updates configurations in the current directory for easy readability and consistency?

- terraform validate
- terraform init
- terraform format
- terraform fmt
  (Correct)
- terraform check

Explanation

Format and validate the configuration

We recommend using consistent formatting in files and modules written by different teams. The terraform fmt command automatically updates configurations in the current directory for easy readability and consistency.

Format your configuration. Terraform will return the names of the files it formatted. If your configuration file is already formatted correctly, Terraform won't return any file names.

```
$ terraform fmt
```

Click here to know more

Question 8: Skipped
How does Infrastructure as Code (IaC) make infrastructure more reliable? Choose *FOUR* correct answers.

- With IaC, we can test the code and review the results before the code is applied to our target environments.
  (Correct)
- Without IaC, an operator might open multiple sessions during infrastructure changes, which often results in skipped steps that can result in slight differences between servers that compound over time and could impact their performance, usability, or security.
  (Correct)
- Scaling up infrastructure becomes more reliable when the operator manually connects to each machine remotely and provision and configure many servers by executing a series of commands/scripts.
- IaC makes changes idempotent, consistent, repeatable, and predictable.
  (Correct)
- IaC automates infrastructure changes. It has no relation to its reliability.
- Since code is checked into version control systems such as GitHub, GitLab, BitBucket, etc., it is possible to review how the infrastructure evolves over time.
  (Correct)

Explanation

IaC Makes Infrastructure More Reliable

IaC makes changes idempotent, consistent, repeatable, and predictable. Without IaC, scaling up infrastructure to meet increased demand may require an operator to remotely connect to each machine and then manually provision and configure many servers by executing a series of commands/scripts. They might open multiple sessions and move between screens, which often results in skipped steps or slight variations between how work is completed, necessitating rollbacks. Perhaps a command was run incorrectly on one instance and reverted before being re-run correctly.

These process inconsistencies can result in slight differences between servers that compound over time and could impact their performance, usability, or security. If a large team is applying changes, the risks increase because individuals don't always follow the same instructions identically.

With IaC, we can test the code and review the results before the code is applied to our target environments. Should a result not align to our expectations, we iterate on the code until the results pass our tests and align to our expectations. Following this pattern allows

for the outcome to be predicted before the code is applied to a production environment. Once ready for use, we can then apply that code via automation, at scale, ensuring consistency and repeatability in how it is applied.

Since code is checked into version control systems such as GitHub, GitLab, BitBucket, etc., it is possible to review how the infrastructure evolves over time. The idempotent characteristic provided by IaC tools ensures that, even if the same code is applied multiple times, the result remains the same.

Click here to know more about it

Question 9: Skipped
Peter Griffin creates two workspaces named dev and prod.

Select all statements that apply to the context. Choose *THREE* correct answers.

- State from each workspace will point to the same resources in the real world.
- Each workspace will point to the same state file terraform.tfstate.
- Peter will have two workspaces named dev and prod.
- Peter will have three workspaces named dev, prod, and default.
  (Correct)
- Peter can delete dev workspace, but he can not delete the default workspace.
  (Correct)
- Each workspace has a separate state file associated with it.
  (Correct)

Explanation

Initially, the backend has only one workspace, called "default", and thus there is only one Terraform state associated with that configuration.

Terraform starts with a single workspace named "default". This workspace is special both because it is the default and also because it cannot ever be deleted. If you've never explicitly used workspaces, then you've only ever worked on the "default" workspace.

Certain backends support *multiple* named workspaces, allowing multiple states to be associated with a single configuration. The configuration still has only one backend, but multiple distinct instances of that configuration to be deployed without configuring a new backend or changing authentication credentials.

When you create a new workspace, Terraform will not see any existing resources that existed on the default (or any other) workspace. **These resources still physically exist,** but are managed in another Terraform workspace.

For local state, Terraform stores the workspace states in a directory called terraform.tfstate.d. This directory should be treated similarly to local-only terraform.tfstate .

Click here to know more about Terraform Workspaces

Question 10: Skipped
Lois used configuration given below and created three users, "Peter", "Chris", and "Stewie", provided with a default value of the user_names variable.

```
variable "user_names" {
description = "Create IAM users"
type        = list(string)
default     = ["Peter", "Chris", "Stewie"]
}

resource "aws_iam_user" "quiz_experts_user" {
count = length(var.user_names)
name  = var.user_names[count.index]
}
```

What changes will occur if Lois changes default value as given below and runs terraform apply command?

```
variable "user_names" {
description = "Create IAM users"
type        = list(string)
default     = ["Peter", "Brian"]
}

resource "aws_iam_user" "quiz_experts_user" {
count = length(var.user_names)
name  = var.user_names[count.index]
}
```

- Terraform will stop managing user "Chris" and "Stewie" and a new user "Brian" will get created.
- User "Chris" and "Stewie" will get deleted, and a new user "Brian" will get created.
-

User "Stewie" will get deleted, and the user "Chris" will get renamed to "Brian".
(Correct)

- User "Chris" and "Stewie" will not get deleted, and a new user "Brian" will get created.

Explanation

When you use the count parameter on a resource, that resource becomes a list of resources. Unfortunately, the way Terraform identifies each resource within the list is by its position (index) in that list. That is, after running terraform apply the first time with three user names, Terraform's internal representation of these IAM users looks something like this:

```
aws_iam_user.quiz_experts_user[0]: Creation complete after 3s [id=Peter]
aws_iam_user.quiz_experts_user[2]: Creation complete after 3s [id=Stewie]
aws_iam_user.quiz_experts_user[1]: Creation complete after 3s [id=Chris]
```

When you remove an item from the list or add new items, all the items after it shifts back or forth, so after running plan with just two user names, Terraform's internal representation will look something like this:

```
aws_iam_user.quiz_experts_user[0]: Refreshing state... [id=Peter]
aws_iam_user.quiz_experts_user[1]: Refreshing state... [id=Brian]
```

Complete terraform apply command output will look like this:

```
$ terraform apply
aws_iam_user.quiz_experts_user[0]: Refreshing state... [id=Peter]
aws_iam_user.quiz_experts_user[1]: Refreshing state... [id=Chris]
aws_iam_user.quiz_experts_user[2]: Refreshing state... [id=Stewie]
An execution plan has been generated and is shown below.
Resource actions are indicated with the following symbols:
  ~ update in-place
  - destroy

Terraform will perform the following actions:

# aws_iam_user.quiz_experts_user[1] will be updated in-place
  ~ resource "aws_iam_user" "quiz_experts_user" {
      arn           = "arn:aws:iam::444300417083:user/Chris"
      force_destroy = false
      id            = "Chris"
    ~ name          = "Chris" -> "Brian"
      path          = "/"
      tags          = {}
      unique_id     = "AIDAWO4S6LA5RLSPBMG45"
    }

# aws_iam_user.quiz_experts_user[2] will be destroyed
  - resource "aws_iam_user" "quiz_experts_user" {
    - arn           = "arn:aws:iam::444300417083:user/Stewie" -> null
    - force_destroy = false -> null
    - id            = "Stewie" -> null
    - name          = "Stewie" -> null
    - path          = "/" -> null
    - tags          = {} -> null
    - unique_id     = "AIDAWO4S6LA53VJPSJCZM" -> null
    }

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

Do you want to perform these actions?
  Terraform will perform the actions described above.
  Only 'yes' will be accepted to approve.

Enter a value: yes

aws_iam_user.quiz_experts_user[2]: Destroying... [id=Stewie]
aws_iam_user.quiz_experts_user[1]: Modifying... [id=Chris]
aws_iam_user.quiz_experts_user[2]: Destruction complete after 3s
aws_iam_user.quiz_experts_user[1]: Modifications complete after 4s [id=Brian]

Apply complete! Resources: 0 added, 1 changed, 1 destroyed.
```

Notice how Terraform modifies index 1 user since Terraform sees the index as a resource's identity, to Terraform, this change roughly translates to *"rename the user at index 1 to Brian and delete the user at index 2."*

Reference

Question 11: Skipped
Which of the following is the primitive type constraint in Terraform? Choose *TWO* correct answers.

- Bool
(Correct)
- Array
- String
(Correct)
- List
- Integer

Explanation

Primitive Types

A *primitive* type is a simple type that isn't made from any other type. All primitive types in

Terraform are represented by a type keyword. The available primitive types are:

* string: a sequence of Unicode characters representing some text, such as "hello".

* number: a numeric value. The number type can represent both whole numbers like 15 and fractional values such as 6.283185.

* bool: either true or false. bool values can be used in conditional logic.

Click here to know more about Type Constraints in Terraform

Question 12: Skipped
When it comes to Terraform 0.12, what does the terraform init command do? Choose *THREE* correct answers.

- Searches the configuration for direct and indirect references to providers and load plugins distributed by HashiCorp.
(Correct)
- Searches the configuration and load plugins only if a valid *provider block* configuration is found.
- Sees the root configuration directory for backend configuration, and initialized the chosen backend.
(Correct)
- Searches for module blocks and the source code for referenced modules is retrieved from the locations given in their source arguments.
(Correct)
- Downloads and installs community as well as HashiCorp distributed provider's plugin so that it can later be executed during terraform plan and apply.
- Downloads and initialize backend and provider; however, module code must pre-exist at the locations given in their source arguments.

Explanation

Plugin Installation

During init, Terraform searches the configuration for both direct and indirect references to providers and attempts to load the required plugins.

Backend Initialization

During init, the root configuration directory is consulted for backend configuration and the chosen backend is initialized using the given configuration settings.

Child Module Installation

During init, the configuration is searched for module blocks, and the source code for referenced modules is retrieved from the locations given in their source arguments.

Community/Third-party Plugins (Terraform 0.12)

Anyone can develop and distribute their own Terraform providers. These third-party providers must be manually installed, since terraform init cannot automatically download them.

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:

Operating system	User plugins directory
Windows	`%APPDATA%\terraform.d\plugins`
All other systems	`~/.terraform.d/plugins`

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.

Providers distributed by HashiCorp can also go in the user plugins directory. If a manually installed version meets the configuration's version constraints, Terraform will use it instead of downloading that provider. This is useful in airgapped environments and when testing pre-release provider builds.

Community/Third-party Plugins (Terraform 0.13) <- *New*

With Terraform 0.13, terraform init will automatically download and install partner and community providers in the HashiCorp Terraform Registry, following the same clear workflow as HashiCorp-supported official providers. These improvements to the ecosystem will benefit Terraform users and provider developers alike.

NOTE: Terraform recently announced Terraform 0.13, it can do the automatic installation of third-party (community) providers.

Hence, in the exam, you can be asked about terraform init command behavior in Terraform 0.13, then you should also mark -> Downloads and installs community as well as HashiCorp distributed provider's plugin so that it can later be executed during terraform plan and apply.

Click here to know more about the terraform init command

New -> Click here to know Automatic Installation of Third-Party Providers with Terraform 0.13

Question 13: Skipped
You have created a new workspace called DEV under directory /home/quiz_experts/iac/ and initialized terraform by running terraform init command with *local* backend. If you run terraform apply , where does Terraform will write its state data?

- /home/quiz_experts/iac/workspace.d/DEV/terraform.tfstate
- /home/quiz_experts/iac/terraform.tfstate.d/terraform.tfstate
- /home/quiz_experts/iac/DEV/terraform.tfstate
- /home/quiz_experts/iac/terraform.tfstate.d/DEV/terraform.tfstate
(Correct)
- /home/quiz_experts/iac/workspace.d/terraform.tfstate
- /home/quiz_experts/iac/terraform.tfstate

Explanation

Workspace

A common use for multiple workspaces is 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. For example, a developer working on a complex set of infrastructure changes might create a new temporary workspace in order to freely experiment with changes without affecting the default workspace.

Workspaces are technically equivalent to renaming your state file. They aren't any more complex than that. Terraform wraps this simple notion with a set of protections and support for remote state.

For local state, Terraform stores the workspace states in a directory called terraform.tfstate.d. This directory should be treated similarly to local-only terraform.tfstate

So for non-default workspaces state will be stored in a file -

terraform.tfstate.d/<your-workspace-name>/terraform.tfstate

In this example our current working directory is /home/quiz_experts/iac/ , hence directory should look like this -

```
.
├── main.tf
├── terraform.tfstate        # Default workspace state file
└── terraform.tfstate.d
   ├── DEV
   │   └── terraform.tfstate  # DEV workspace state file
   └── PROD
       └── terraform.tfstate  # PROD workspace state file
```

For a workspace named DEV , state file will be - /home/quiz_experts/iac/terraform.tfstate.d/DEV/terraform.tfstate

Click here to more about workspaces

Question 14: Skipped
How to add provider to Terraform v0.12 configuration?

- Add any resource from the required provider and Terraform will automatically download the provider while running terraform init command.
- Both of these
(Correct)
- None of these
- Explicitly add provider configuration using a provider block.

Explanation

A new provider can be added to a configuration -- either explicitly via a provider block or by adding a resource from that provider -- Terraform must initialize it before it can be used. Initialization downloads and installs the provider's plugin so that it can later be executed.

terraform init command will download and initialize any providers that are not already initialized.

NOTE: In Terraform v0.12 terraform init command cannot automatically download Third-party Plugins providers that are not distributed by HashiCorp, but in Terraform v0.13 terraform init command can do it automatically.

Terraform v0.12 or earlier	Cannot automatically download third-party providers.
Terraform v0.13 or later	Can automatically download third-party providers.

Click here to see the terraform init command

Click here to know more about old the terraform init command

Click here to see what is new in Terraform v0.13

Question 15: Skipped

What are the advantages of using Infrastructure as Code? Choose *FIVE* correct answers.

- Infrastructure as Code is turning the Ops team into developers.
- Infrastructure as Code codifies manual pointing and clicking on management console and automate it.
  (Correct)
- Infrastructure as Code provides version-control of infrastructure configuration to provide a single source of truth.
  (Correct)
- Infrastructure as Code can be an essential part of your disaster recovery strategy. It helps you stand up new infrastructure quickly and efficiently.
  (Correct)
- Infrastructure as Code provides the ability to gracefully roll the infrastructure back to a last known good state.
  (Correct)
- With Infrastructure as Code, users can write configuration once and then reuse it many times.
  (Correct)

Explanation

Infrastructure as Code

Infrastructure as Code is the process of managing infrastructure in a file or files rather than manually configuring resources in a user interface. A resource in this instance is any piece of infrastructure in a given environment, such as a virtual machine, security group, network interface, etc.

Infrastructure as Code (IaC) is the management of infrastructure (networks, virtual machines, load balancers, and connection topology) in a descriptive model, using the same versioning as DevOps team uses for source code. Like the principle that the same source code generates the same binary, an IaC model generates the same environment every time it is applied. IaC is a key DevOps practice and is used in conjunction with Continuous Delivery.

At a high level, Terraform allows operators to use HCL to author files containing definitions of their desired resources on almost any provider (AWS, GCP, GitHub, Docker, etc) and automates the creation of those resources at the time of apply.

Benefits of infrastructure as code

Infrastructure as code brings a lot of benefits -

Visibility: An infrastructure as code template serves as a very clear reference of what resources are on your account, and what their settings are. You don't have to navigate to the web console to check the parameters.

Stability: If you accidentally change the wrong setting or delete the wrong resource in the web console you can break things. Infrastructure as code helps solve this, especially when it is combined with version control, such as Git.

Scalability: With infrastructure as code you can write it once and then reuse it many times. This means that one well-written template can be used as the basis for multiple services, in multiple regions around the world, making it much easier to horizontally scale.

Security: Once again infrastructure as code gives you a unified template for how to deploy your architecture. If you create one well-secured architecture you can reuse it multiple times, and know that each deployed version is following the same settings.

Transactional: Terraform not only creates resources on your cloud account but also waits for them to stabilize while they start. It verifies that provisioning was successful, and if there is a failure it throws an error and try to recreate resource again during next apply.

Click here to see a video to know what Infrastructure as Code is, and why is it important?

Click here to know more about Infrastructure as Code

Question 16: Skipped

Refer to the configuration block given below and suggest the value of X to get us-east-2 as *output* selected_region.

```
variable "region_list" {
description = "List of AWS regions"
type       = list(string)
default    = ["us-east-1", "us-east-2", "us-west-1"]
}
```

```
output "selected_region" {
value = var.region_list[X]
}
```

- 2
- "us-east-2"
- None of these
- 1
  (Correct)

Explanation

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

Since the list index starts from zero, here are the output values for corresponding X values -

X = 0	var.region_list[0]	"us-east-1"
X = 1	var.region_list[1]	"us-east-2"
X = 2	var.region_list[2]	"us-west-1"

```
variable "region_list" {
  description = "List of AWS regions"
  type        = list(string)
  default     = ["us-east-1", "us-east-2", "us-west-1"]
}
output "selected_region" {
  value = var.region_list[1]
}
```

Question 17: Skipped
Suppose you manage resources such as databases with Terraform similar to the example code given below. Select all security measures you must take to protect sensitive data such as username and password? Choose *THREE* correct answers.

```
resource "aws_db_instance" "quiz_experts_db" {
engine            = "mysql"
engine_version    = "5.7"
instance_class    = "db.t2.micro"
name              = "quiz_experts_db"

# Set the secrets from variables
username = var.username
password = var.password
}
```

- Storing state remotely can provide better security.
  (Correct)
- Use local backend as it always encrypt state files in memory before storing it to files.
- Local state provides better security as state data never leaves your system.
- Strictly control who can access your Terraform backend.
  (Correct)
- If you manage any sensitive data with Terraform (like database passwords, user passwords, or private keys), treat the state itself as sensitive data, always encrypt state at rest, and protect it with TLS in transit.
  (Correct)
- Use *sensitive* argument to stop sensitive values from being recorded in the state.

Explanation

Sensitive Data in State

Terraform state can contain sensitive data, depending on the resources in use and your definition of "sensitive." The state contains resource IDs and all resource attributes. For resources such as databases, this may contain initial passwords.

When using local state, state is stored in plain-text JSON files.

When using remote state, state is only ever held in memory when used by Terraform. It may be encrypted at rest, but this depends on the specific remote state backend.

Recommendations

If you manage any sensitive data with Terraform (like database passwords, user passwords, or private keys), treat the state itself as sensitive data.

Storing state remotely can provide better security. As of Terraform 0.9, Terraform does not persist state to the local disk when remote state is in use, and some backends can be configured to encrypt the state data at rest.

Click here to know more about Terraform Recommendations for Sensitive Data

Question 18: Skipped
Let's say you are using AWS provider, and you need to create two EC2 instances in two different AWS regions. Is it possible to create resources in more than one region with a single configuration?

-

Yes, include multiple provider blocks with the same provider name. For each additional non-default configuration, use the alias meta-argument to provide a new name segment.
(Correct)

- Yes, include multiple provider blocks with the different provider name.
- Yes, region argument under provider blocks supports list.

```
provider "aws" {
region = [ "us-east-1" , "us-west-2" ]
}
```

- Yes, we need to define two provider block one for each region as given below -

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

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

- No, You need to use multiple configurations for the same provider to create resources in different regions.

Explanation

Multiple Provider Configurations

You can optionally define multiple configurations for the same provider, and select which one to use on a per-resource or per-module basis. The primary reason for this is to support multiple regions for a cloud platform; other examples include targeting multiple Docker hosts, multiple Consul hosts, etc.

To create multiple configurations for a given provider, include multiple provider blocks with the same provider name. For each additional non-default configuration, use the alias meta-argument to provide an extra name segment. For example:

```
# The default provider configuration; resources that begin with `aws_` will use
# it as the default, and it can be referenced as `aws`.
provider "aws" {
  region = "us-east-1"
}
# Additional provider configuration for west coast region; resources can
# reference this as `aws.west`.
provider "aws" {
  alias  = "west"
  region = "us-west-2"
}
```

Selecting Alternate Provider Configurations

By default, resources use a default provider configuration (one without an alias argument) inferred from the first word of the resource type name.

To use an alternate provider configuration for a resource or data source, set its provider meta-argument to a <PROVIDER NAME>.<ALIAS> reference:

```
resource "aws_instance" "foo" {
  provider = aws.west
# ...
}
```

Click here to know more about Multiple Provider Configurations

Question 19: Skipped
What provisioner can you use to that invokes a command on the machine that runs Terraform CLI?

- exec-local
- local-exec
(Correct)
- remote-exec
- local-cmd
- salt-masterless

Explanation

local-exec Provisioner

The local-exec provisioner invokes a local executable after a resource is created. This invokes a process on the machine running Terraform, not on the resource.

Example:

```
resource "aws_instance" "quiz_experts" {
  # ...
provisioner "local-exec" {
    command = "echo ${aws_instance.quiz_experts.private_ip} >> private_ips.txt"
  }
}
```

Note: Provisioners should only be used as a last resort.

Click here to know more about local-exec provisioner

Question 20: Skipped

Your manager has asked you to create a strong separation between development and production deployments to ensure users will only touch the intended infrastructure. Any change in one deployment should not impact another deployment. Which of the following, you think is the right approach?

- Modify Terraform configuration and use different resource nomenclature for different deployments.
- Use a deployment name as input variable and create a condition in each resource block to set count 0 or 1 based on deployment name.
- Use separate remote backend storage for each deployment.
- Use separate Terraform workspaces for each deployment as each workspace has a different state file associated with it.
- Use separate working-directories for each deployment. Each working directory should have its separate deployment-specific configuration and backend with different credentials and access controls.
  (Correct)

Explanation

Named workspaces allow conveniently switching between multiple instances of a *single* configuration within its *single* backend. They are convenient in a number of situations, but cannot solve all problems.

A common use for multiple workspaces is 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. For example, a developer working on a complex set of infrastructure changes might create a new temporary workspace in order to freely experiment with changes without affecting the default workspace.

Organizations commonly want to create a strong separation between multiple deployments of the same infrastructure serving different development stages (e.g. staging vs. production) or different internal teams. In this case, the backend used for each deployment often belongs to that deployment, with different credentials and access controls. Named workspaces are *not* a suitable isolation mechanism for this scenario because each subsystem should have its own separate configuration and backend, and will thus have its own distinct set of workspaces.

```
$ tree
.
├── assets
│   ├── index.html
├── prod  # Separate directory for production deployment
│   ├── main.tf
│   ├── variables.tf
│   ├── terraform.tfstate
│   └── terraform.tfvars
└── dev   # Separate directory for development deployment
    ├── main.tf
    ├── variables.tf
    ├── terraform.tfstate
    └── terraform.tfvars
```

Click here to know more

Question 21: Skipped

You created three VMs for an application using a Terraform configuration, as illustrated in this terraform apply output:

```
aws_instance.quiz_experts[0]: Creation complete after 42s [id=i-0fe5dc9f90a5c9376]
aws_instance.quiz_experts[1]: Creation complete after 45s [id=i-06240e6517a27559d]
aws_instance.quiz_experts[2]: Creation complete after 42s [id=i-00eb7ce0eed4df485]
```

Now because of some direct manual changes on aws_instance.quiz_experts[1], you want to re-create just one aws_instance.quiz_experts[1] machine. How can you do it?

- Run below given two commands -
  terraform destroy -auto-approve
  terraform apply -auto-approve
- Run below given two commands -
  terraform taint "aws_instance.quiz_experts[1]"
  terraform apply
  (Correct)
- Run command -
  terraform recreate -target="aws_instance.quiz_experts[1]"
- Run below given two commands -
  terraform refresh -target="aws_instance.quiz_experts[1]"
  terraform apply
- Run command -
  terraform taint "aws_instance.quiz_experts[1]"

Explanation

Manually Tainting Resources

In cases where you want to manually destroy and recreate a resource, Terraform has a built-in taint function in the CLI. 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.

To taint a resource, use the following command:

```
$ terraform taint resource.id
```

resource.id refers to the resource block name and resource ID to taint.

```
resource "aws_instance" "quiz_experts_vm" {
  ami           = "ami-fc0a942c"
  instance_type = "t2.micro"
}
```

The correct resource and ID to taint this resource given in the above configuration would be:

```
terraform taint aws_instance.quiz_experts_vm
```

Hence taint is the preferred way to do this using built-in re-creation features is to mark the resource as tainted and then to rerun the terraform apply process. Based on terraform apply the output given in the question we need to taint *index* 1 instance from the group of instances created by one resource block:

```
terraform taint "aws_instance.quiz_experts[1]"
terraform apply
```

You could also do this by using a targeted terraform destroy:

```
terraform destroy -target="aws_instance.quiz_experts[1]"
terraform apply
```

Click here to know more about resource taint

Question 22: Skipped
Remote state allows teams to share infrastructure resources in a read-only way without relying on any additional configuration store.

- False
- True
  (Correct)

Explanation

Delegation and Teamwork with Remote State

Remote state gives you more than just easier version control and safer storage. It also allows you to delegate the outputs to other teams. This allows your infrastructure to be more easily broken down into components that multiple teams can access.

Put another way, remote state also allows teams to share infrastructure resources in a read-only way without relying on any additional configuration store.

For example, a core infrastructure team can handle building the core machines, networking, etc. and can expose some information to other teams to run their own infrastructure. As a more specific example with AWS: you can expose things such as VPC IDs, subnets, NAT instance IDs, etc. through remote state and have other Terraform states consume that.

For example usage, see the terraform_remote_state data source.

While remote state is a convenient, built-in mechanism for sharing data between configurations, it is also possible to use more general stores to pass settings both to other configurations and to other consumers. For example, if your environment has HashiCorp Consul then you can have one Terraform configuration that writes to Consul using consul_key_prefix and then another that consumes those values using the consul_keys data source.

Click here to know more

Question 23: Skipped
The Terraform Registry is a public registry. For private use of such a browsable marketplace of modules, you must use the Private Registry available in the Terraform Cloud or Enterprise version.

- False
- True
  (Correct)

Explanation

If you are a customer of HashiCorp's Terraform Enterprise/ Terraform Cloud, you can have this same experience with a Private Terraform Registry; that is, a registry that lives in your private Git repos, and is only accessible to your team. This can be a great way to share modules within your company.

The Terraform open source project does not provide a server implementation, but

community members can create their own private registries by following the published protocol.

Click here to know more about Private Registry

Click here to know more

Question 24: Skipped
Which of the following configuration block has Terraform 0.12 recommended way of provider version constraint?

```
# Terraform Block

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

```
# Provider Block

provider "aws" {
version = "~> 3.0"
}
```

- Terraform 0.12 only supports the required_provider within terraform block, and version is not a valid argument in provider blocks.
- Terraform 0.12 recommend using the version in provider blocks, and *not* using the required_providers argument within terraform block. required_providers is still supported for compatibility with older Terraform versions.
- Terraform 0.12 recommend using the required_providers in terraform block, and *not* using the version argument within provider blocks. version is still supported for compatibility with older Terraform versions.
  (Correct)
- Terraform 0.12 only supports the version within provider blocks, and required_providers is not a valid argument in terraform block.

Explanation

Provider version constraints can also be specified using a version argument within a provider block, but that simultaneously declares a new provider configuration that may cause problems particularly when writing shared modules. For that reason, we recommend using the required_providers block as described above, and *not* using the version argument within provider blocks. version is still supported for compatibility with older Terraform versions.

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

# Only supported for compatibility with older Terraform versions
provider "aws" {
  version = "~> 3.0"
}
```

The version meta-argument specifies a version constraint for a provider, and works the same way as the version argument in a required_providers block. The version constraint in a provider configuration is only used if required_providers does not include one for that provider.

Terraform does not recommend using the version argument in provider configurations. In Terraform 0.13 and later, version constraints should always be declared in the required_providers block.

Click here to know more about Provider Versions

Note: The version meta-argument made sense before ***Terraform 0.13***, since Terraform could only install providers that were distributed by HashiCorp. Now that Terraform can install providers from multiple sources, it makes more sense to keep version constraints and provider source addresses together.

Update: **New in Terraform 0.13**

The name = { source, version } syntax for required_providers was added in Terraform 0.13. Previous versions of Terraform used a version constraint string instead of an object (like mycloud = "~> 1.0"), and had no way to specify provider source addresses. If you want to write a module that works with both Terraform v0.12 and v0.13, see v0.12-Compatible Provider Requirements below.

Now, a provider requirement consists of a local name, a source location, and a version constraint:

```
terraform {
  required_providers {
    mycloud = {
      source  = "mycorp/mycloud"
      version = "~> 1.0"
    }
  }
}
```

Each argument in the required_providers block enables one provider. The key determines the provider's local name (its unique identifier within this module), and the value is an object with the following elements:

* source - the global source address for the provider you intend to use, such as hashicorp/aws.

* version - a version constraint specifying which subset of available provider versions the module is compatible with.

Question 25: Skipped
Terraform cannot reason about what the provisioner does; hence if a creation-time provisioner fails for a resource, Terraform will plan to destroy and recreate resource upon the next terraform apply.

- True
  (Correct)
- False

Explanation

Creation-Time Provisioners

By default, provisioners run when the resource they are defined within is created. Creation-time provisioners are only run during *creation*, not during updating or any other lifecycle. They are meant as a means to perform bootstrapping of a system.

If a creation-time provisioner fails, the resource is marked as **tainted**. A tainted resource will be planned for destruction and recreation upon the next terraform apply. Terraform does this because a failed provisioner can leave a resource in a semi-configured state. Because Terraform cannot reason about what the provisioner does, the only way to ensure proper creation of a resource is to recreate it. This is tainting.

You can change this behavior by setting the on_failure attribute, which is covered in detail below.

Click here to know more about Creation-Time Provisioner

Question 26: Skipped
You can set variables declared in the root module using CLI options and environment variables. But to set variables declared in child modules, you should pass their values in the module block.

- True
  (Correct)
- False

Explanation

When you declare variables in the root module of your configuration, you can set their values using CLI options and environment variables. When you declare them in child modules, the calling module should pass values in the module block.

Assigning Values to Root Module Variables

When variables are declared in the root module of your configuration, they can be set in a number of ways:

* In a Terraform Cloud workspace.

* Individually, with the -var command-line option.

* In variable definitions (.tfvars) files, either specified on the command line or automatically loaded.

* As environment variables.

Assigning Values to Child Module Variables

To *call* a module means to include the contents of that module into the configuration with specific values for its input variables. Modules are called from within other modules using module blocks:

```
module "servers" {
  source = "./app-cluster"
# servers and image_id are child module input variables
  servers  = 5
  image_id = "ami-eijo28"
}
```

Click here to know more about assigning values to root module variables

Question 27: Skipped
The default setting of Terraform Open Source is to store your state file on your local

laptop or your workstation.

- False
- True
  (Correct)

Explanation

By default, Terraform states are stored in a local file named terraform.tfstate under the current working directory, but it can also be stored remotely, which works better in a team environment.

Click here to know more about Terraform State

Question 28: Skipped
Which of the following statements are true for below given Terraform configuration? Choose *TWO* correct answers.

```
# Create virtual network
resource "azurerm_virtual_network" "vnet" {
name              = "QuizExpertsVnet"
address_space      = ["10.0.0.0/16"]
location          = "westus2"
resource_group_name = azurerm_resource_group.rg.name
}

## ...more blocks here...

# Create a resource group
resource "azurerm_resource_group" "rg" {
name     = "QuizExpertsResourceGroup"
location = "westus2"
}
```

- If resource blocks and expressions in the configuration are in random order, use the depends_on *meta-argument* to define these dependencies explicitly.
- This configuration will fail while creating a new Azure VNet. You have to specify the Resource Group's name to contain the VNet. The value of resource_group_name in this configuration is an expression azurerm_resource_group.rg.name and will not be available as you are creating Resource Group after Azure VNet.
- Terraform can infer when one resource depends on another by analyzing the resource attributes used in interpolation expressions. From this, Terraform builds a dependency tree to determine the correct order to create each resource.
  (Correct)
- The order of blocks and expressions in the Terraform configuration file doesn't matter.
  (Correct)

Explanation

Implicit dependencies

Terraform can infer when one resource depends on another by analyzing the resource attributes used in interpolation expressions. The act of referencing an attribute from another resource in the argument value of a resource results in an implicit dependency. From this, Terraform builds a dependency tree to determine the correct order in which to create each resource. Use implicit dependencies wherever possible.

In the question of an implicit dependency, Terraform knows that the azurerm_resource_group has to be created before the azurerm_virtual_network because the virtual network references the resource group in its resource_group_name argument. The expression azurerm_resource_group.rg.name creates an implicit dependency on the azurerm_resource_group object named rg.

```
# Create a resource group
resource "azurerm_resource_group" "rg" {
   name     = "QuizExpertsResourceGroup"
   location = "westus2"
}
# Create virtual network
resource "azurerm_virtual_network" "vnet" {
   name              = "QuizExpertsVnet"
   address_space      = ["10.0.0.0/16"]
   location          = "westus2"
   resource_group_name = azurerm_resource_group.rg.name
}
```

Because of the dependency, the order of blocks and expressions in the configuration file doesn't matter. This example works just like the one above:

```
# Create virtual network
resource "azurerm_virtual_network" "vnet" {
   name              = "QuizExpertsVnet"
   address_space      = ["10.0.0.0/16"]
   location          = "westus2"
   resource_group_name = azurerm_resource_group.rg.name
}
## ...more blocks here...

# Create a resource group
resource "azurerm_resource_group" "rg" {
   name     = "QuizExpertsResourceGroup"
   location = "westus2"
```

}

Explicit dependencies

Sometimes there are dependencies between resources that are not visible to Terraform. In these scenarios, use the depends_on *meta-argument* to explicitly define these dependencies. The depends_on meta-argument accepts a list of resources.

Click here to know more about Terraform Resource Dependencies

Question 29: Skipped
Refer to the example.tf file given below and select the answer that applies. If you run terraform apply three times, how many IP records will you see in the file ip_address.txt?

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

resource "aws_instance" "quiz_experts" {
ami          = "ami-05794d4ef61b053f2"
instance_type = "t2.micro"

provisioner "local-exec" {
command = "echo ${aws_instance.quiz_experts.public_ip} >> ip_address.txt"
}
}
```

- 0
- 1
  (Correct)
- 3
- None of these

Explanation

Creation-Time Provisioners

By default, provisioners run when the resource they are defined within is created. Creation-time provisioners are only run during *creation*, not during updating or any other lifecycle. They are meant as a means to perform bootstrapping of a system. Hence no matter how many times you run creation-time (default) provisioners after creating a resourse it will not run.

```
% cat ip_address.txt
23.20.23.157
```

Click here to know more about Creation-Time Provisioners

Question 30: Skipped
A calling module can directly access child module resource attributes.

- False
  (Correct)
- True

Explanation

The resources defined in a module are encapsulated, so the calling module cannot access their attributes directly. However, the child module can declare output values to selectively export certain values to be accessed by the calling module.

For example, if the ./app-cluster module referenced in the example above exported an output value named instance_ids then the calling module can reference that result using the expression module.instances.instance_ids:

```
resource "aws_elb" "quiz_experts_elb" {
  # ...
instances = module.instances.instance_ids
}
```

Click here to know more about Accessing Module Output Values

Question 31: Skipped
Sentinel is a _____ service that prevents the provisioning of out-of-policy infrastructure.

- Reactive
- Passive
- Proactive
  (Correct)
- Prescriptive
- Prescribed

Explanation

Sentinel Policy as Code

Sentinel is a policy as a code framework that's integrated into Hashicorp enterprise products.

Compared to many tools that scan existing infrastructure for policy infractions, Sentinel proactively prevents provisioning of out-of-policy infrastructure. Policy as code can be written to achieve a range of goals from limiting oversized cloud deploys to ensuring all infrastructure is properly tagged.

Sentinel allows users to define policies that are enforced against infrastructure between the plan and apply phases of a Terraform run.

Click here to know more about Sentinel

Question 32: Skipped
In regards to Terraform Cloud, Select all correct statements. Choose *TWO* correct answers.

- Terraform Cloud is self-hosted distribution of Terraform.
- Terraform Cloud 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.
  (Correct)
- Workspaces in open-sourced Terraform and Terraform Cloud are identical.
- Terraform Cloud always encrypts state at rest and protects it with TLS in transit.
  (Correct)

Explanation

Terraform Cloud

Terraform Cloud is an application that helps teams use Terraform together. It manages Terraform runs in a consistent and reliable environment, and includes easy access to shared state and secret data, access controls for approving changes to infrastructure, a private registry for sharing Terraform modules, detailed policy controls for governing the contents of Terraform configurations, and more.

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.

Click here to know more about Terraform Cloud

Question 33: Skipped
What does the below-given command do?

495. **export TF_LOG=**

- Does nothing.
- Enable detailed logs to appear on screen.
- Disable detailed logs.
  (Correct)
- Enable detailed logs to the file set in environment variable TF_LOG_PATH.

Explanation

To disable, either unset it or set it to empty.

```
# Set it to empty
export TF_LOG=
```

OR

```
# Unset it
unset TF_LOG
```

Click here to know more about TF_LOG

Question 34: Skipped
A Terraform _____ determines how Terraform loads and stores state.

- Resource
- Backend
  (Correct)
- Provider
- Data Block

Explanation

A "**backend**" in Terraform determines how state is loaded and how an operation such

as apply is executed.

By default, Terraform uses the "local" backend, which stores the state file on your local disk. Remote backends allow you to store the state file in a remote, shared store.

Here are some of 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.

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.

Even if you only intend to use the "local" backend, it may be useful to learn about backends since you can also change the behavior of the local backend.

Click here to know more about Terraform Backends

Question 35: Skipped

Let's say a data center went down completely, and we need to failover to a different datacenter or a cloud environment. How does Terraform can help in this scenario? Choose *TWO* correct answers.

- Terraform allows cheaper and much more dynamic disaster recovery plans because you can literally build a DR environment on the fly; hence it may not make sense to keep it running all the time and pay a massive bill for it.
(Correct)
- Terraform backs up the data and use various strategies to help you restore databases and everything you need to keep your environments up and running during disaster recovery.
- During disaster recovery, Terraform can help you very quickly stand up a new infrastructure or new production environments in a different region of the same cloud provider in a very short period of time; however, Terraform may charge a small amount for a cross-cloud provider DR.
- Terraform can be an essential part of your disaster recovery strategy because it helps you stand up new infrastructure very quickly and efficiently.
(Correct)

Explanation

Disaster Recovery with Terraform

Let's say the datacenter went completely down and we need to failover to a different datacenter or a cloud environment. You can use Terraform to very quickly stand up all the application infrastructure and get everything ready. So Terraform can be a core piece of your DR recovery plans.

One thing that Terraform won't do, though, is back up your data, so it's very important that you have a good backup strategy to use along with Terraform so that, once your servers and applications are up and running, you also have the databases and everything that you need to keep your environments up and running.

Practice makes perfect, even with DR

Another thing to keep in mind is that a DR plan without practice is not really a DR plan. You should be doing regular tests of your DR strategy just to make sure that, when things really go wrong, you know it will work because you tested it last week or last month. Terraform makes this very easy. You can use Terraform to stand up copies of your environment and do practice runs, like a fire drill. Make sure that your DR plans work before the disaster happens and not after.

Terraform can be an essential part of your disaster recovery strategy because it helps you stand up new infrastructure very quickly and efficiently.

Click here to know more about DR with Terraform

Question 36: Skipped

You created a Terraform module and pushed it to your company's private git repository. You recently observed that this module has a bug; hence, you checked out a new hotfix branch from the master branch and fixed it. How can you test the new module code before merging it to the master?

- Run a test configuration by changing the module source to the value given below and make sure it works as expected.

```
module "vpc" {
source = "git::https://code.quizexperts.com/vpc.git"
```

```
version = "hotfix"
}
```

- Run a test configuration by changing the module source to the value given below and make sure it works as expected.

```
module "vpc" {
source = "git::https://code.quizexperts.com/vpc.git#ref=hotfix"
}
```

- Run a test configuration by changing the module source to the value given below and make sure it works as expected.

```
module "vpc" {
source = "git::https://code.quizexperts.com/vpc.git?ref=hotfix"
}
```

(Correct)

- Run a test configuration by changing the module source to the value given below and make sure it works as expected.

```
module "vpc" {
source  = "git::https://code.quizexperts.com/vpc.git"
branch = "hotfix"
}
```

- Run a test configuration by changing the module source to the value given below and make sure it works as expected.

```
module "vpc" {
source = "git::https://code.quizexperts.com/vpc.git?branch=hotfix"
}
```

- Run a test configuration by changing the module source to the value given below and make sure it works as expected.

```
module "vpc" {
source = "git::https://code.quizexperts.com/vpc.git&branch=hotfix"
}
```

Explanation

Selecting a Revision

By default, Terraform will clone and use the default branch (referenced by HEAD) in the selected repository. You can override this using the ref argument:

```
# To specify a Tag
module "vpc" {
  source = "git::https://code.quizexperts.com/vpc.git?ref=v1.2.0"
}


# To specify a Branch
module "vpc" {
  source = "git::https://code.quizexperts.com/vpc.git?ref=hotfix"
}
```

You have a generic ref there that Git will work out what you mean by context assuming no collisions.

If you do have 2 refs that are ambiguous then Git will error and tell you that it's an ambiguous ref and force you to specify the full ref using refs/heads/branch-name or refs/tags/tag-name.

The value of the ref argument can be any reference that would be accepted by the git checkout command, including branch and tag names.

Click here to know more about Selecting a revision

Click here to know more about such cases on stackoverflow.com

NOTE: This is an important topic for the exam.

Question 37: Skipped
Version constraints are supported only for modules installed from a module registry.

- True
(Correct)
- False

Explanation

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

Click here to know more about Module Versions

Question 38: Skipped
The parent module has to provide input variables and providers to a child module explicitly in the module block as this information can not be passed down to descendent modules implicitly through inheritance.

- False
(Correct)
-

True

Explanation

Providers

- Providers can be passed down to descendent modules in two ways: either *implicitly* through inheritance, or *explicitly* via the providers argument within a module block. These two options are discussed in more detail in the following sections.

- Provider configurations can be defined only in a root Terraform module.

- A module intended to be called by one or more other modules must not contain any provider blocks, with the exception of the special "proxy provider blocks"

- Additional provider configurations (those with the alias argument set) are *never* inherited automatically by child modules, and so must always be passed explicitly using the providers map.

Example:

```
provider "aws" {
  alias  = "usw1"
  region = "us-west-1"
}
provider "aws" {
  alias  = "usw2"
  region = "us-west-2"
}

module "tunnel" {
  source    = "./tunnel"
  providers = {
    aws.src = aws.usw1
    aws.dst = aws.usw2
  }
}
```

Input Variables

The parent has to provide input variables values to the child module explicitly in the module block.

Example:

```
module "aws_network_vpc" {
  source           = "./modules/vpc"
  aws_vpc_block    = var.aws_vpc_block
  aws_vpc_tag_name = var.aws_vpc_tag_name
}
```

Click here to know more about Providers in Modules

Click here to know more about Calling a Child Module

Click here to know about Proxy Configuration Blocks

Question 39: Skipped

You are working on a new load balancer module, and your boss asked you to leverage an already created instance module written by a different team. How can you source this module from a separate private git repository?

-

```
516. module "instances" {
517. source = "https://code.quizexperts.com/instances.git"
518. }
```

-

```
519. module "instances" {
520. source = "ssh://username@code.quizexperts.com:instances.git"
521. }
```

-

```
522. module "instances" {
523. source = "git::ssh://username@code.quizexperts.com/instances.git"
524. }
```

(Correct)

-

```
525. module "instances" {
526. source = "git@code.quizexperts.com/terraform/instances.git"
527. }
```

Explanation

Generic Git Repository

Arbitrary Git repositories can be used by prefixing the address with the special git:: prefix. After this prefix, any valid Git URL can be specified to select one of the protocols supported by Git.

For example, to use HTTPS or SSH:

```
module "vpc" {
  source = "git::https://example.com/vpc.git"
}
module "storage" {
  source = "git::ssh://username@example.com/storage.git"
```

```
}
```

Terraform installs modules from Git repositories by running git clone, and so it will respect any local Git configuration set on your system, including credentials. To access a non-public Git repository, configure Git with suitable credentials for that repository.

If you use the SSH protocol then any configured SSH keys will be used automatically. This is the most common way to access non-public Git repositories from automated systems because it is easy to configure and allows access to private repositories without interactive prompts.

If using the HTTP/HTTPS protocol, or any other protocol that uses username/password credentials, configure Git Credentials Storage to select a suitable source of credentials for your environment.

If your Terraform configuration will be used within Terraform Cloud, only SSH key authentication is supported, and keys can be configured on a per-workspace basis.

Click here to know more about sourcing modules from a private git repo

GitHub

Terraform will recognize unprefixed github.com URLs and interpret them automatically as Git repository sources.

```
module "consul" {
  source = "github.com/hashicorp/example"
}
```

The above address scheme will clone over HTTPS. To clone over SSH, use the following form:

```
module "consul" {
  source = "git@github.com:hashicorp/example.git"
}
```

Click here to know more about sourcing modules from GitHub

Question 40: Skipped

A variable is defined by using a *variable* block with a label, which of the following is a valid variable block?

-

```
variable "count" {
type        = "number"
description = "Number of resources to launch"
}
```

-

```
variable "count" {
type        = number
description = "Number of resources to launch"
}
```

-

```
variable "counts" {
type        = number
description = "Number of resources to launch"
}
```

(Correct)

-

```
variable "counts" {
type        = integer
description = "Number of resources to launch"
}
```

Explanation

The label after the variable keyword is a name for the variable, which must be unique among all variables in the same module. This name is used to assign a value to the variable from outside and to reference the variable's value from within the module.

The name of a variable can be any valid identifier except the following:

- `source`
- `version`
- `providers`
- `count`
- `for_each`
- `lifecycle`
- `depends_on`
- `locals`

These names are reserved for meta-arguments in module configuration blocks, and cannot be declared as variable names.

The variable declaration can optionally include a type argument to specify what value types are accepted for the variable, as described in the following section.

The variable declaration can also include a default argument. If present, the variable is considered to be *optional* and the default value will be used if no value is set when calling the module or running Terraform. The default argument requires a literal value and cannot reference other objects in the configuration.

Click here to more about Variables

Question 41: Skipped
Terraform recommends using provisioners instead of configuration management tools to invoke scripts or installing software on a newly built VMs.

- False
(Correct)
- True

Explanation

Terraform Provisioners - "*Provisioners are a Last Resort*"

Provisioners should only be used as a last resort. For most common situations there are better alternatives like user_data in AWS EC2 or tools like Chef/Ansible/Saltstack built specifically for configuration management.

Click here to know more

Question 42: Skipped
You have started getting lots of conflicts and inconsistencies in the infrastructure managed by Terraform ever since You began using Amazon S3 as Terraform backend. After some troubleshooting, you observed that this only happens when more than one user tries to make infrastructure changes with Terraform. How will you resolve it?

- Enable Terraform state locking for the Amazon S3 backend by using a DynamoDB table to avoid race around the condition and concurrent execution problem of the same Terraform configuration.
(Correct)
- Ask users to notify all team members if a user needs to run Terraform CLI so that no other user runs Terraform CLI at the same time.
- Open-source Terraform only allows one user to use an S3 backend. You need to upgrade to Terraform Cloud or Enterprise to enable multiple users to work on the same state concurrently.
- Add rate limiting rule on Amazon S3 bucket to accept only one request for a duration of time.
- Use Terraform workspace as each workspace has a separate state file that resolves conflicts and inconsistencies problem of shared states.
- Add outbound rate limit in ip-tables of the server running terraform CLI.

Explanation

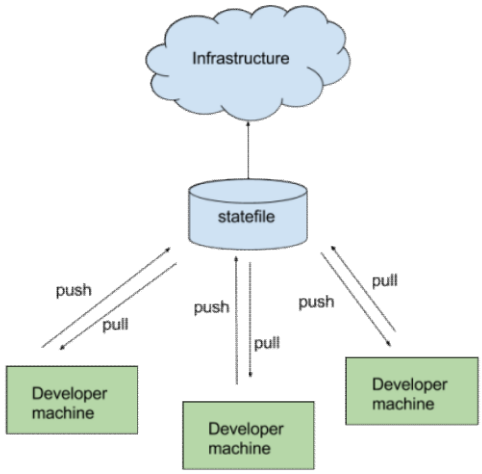


Image source - Internet

Why state locking?

Storing the state remotely lets you collaborate smoothly. It allows each of your team members to access the latest state files, but it also brings a risk of accidental state file corruption, especially when two team members are running Terraform at the same time you may run into race conditions as multiple terraform apply commands make concurrent updates to the same state files, leading to conflicts and data loss, and state file corruption.

State lock, a feature that prevents opening a state file while already in use, will resolve this problem. Please note that not all backends support locking. If your backend supports it and is enabled, Terraform will lock your state for all operations that could write state. This prevents others from acquiring the lock and potentially corrupting your state.

NOTE: Not all backends support locking. Please view the list of backend types for details on whether a backend supports locking or not.

You can implement the lock by creating an AWS DynamoDB Table.

Image source - Internet

To use DynamoDB for locking with Terraform, you must create a DynamoDB table with a primary key called LockID with this exact spelling and capitalization.

Here's what the sample backend configuration looks like for an S3 backend with state lock:

```
terraform {
  backend "s3" {
    # Replace this with your bucket name.
    bucket         = "quiz-experts-terraform-bucket"
    key            = "global/quiz-experts/stats/terraform.tfstate"
    region         = "us-east-2"
# Replace this with your DynamoDB table name.
    dynamodb_table = "quiz-experts-terraform-locks"
    encrypt        = true
  }
}
```

Remember to run terraform init again after making any change in the remote backend configuration. This command can not only download provider code, but also configure your Terraform backend. Moreover, the init command is *idempotent*, so it's safe to run it over and over again.

Click here to know more about S3 Backend

Click here to know more about State Locking

Question 43: Skipped

To publish a module to Terraform Public Registry. A module must meet the following requirements.

Choose *TWO* correct answers.

- Module GitHub repositories must use this three-part name format **terraform-<PROVIDER>-<NAME>** where <NAME> segment can not contain any special characters.
- The module must be on a public GitHub repo with at least one release tag in x.y.z format.
  (Correct)
- Module GitHub repositories must use this three-part name format **terraform-<PROVIDER>-<NAME>** where <NAME> segment can contain additional hyphens.
  (Correct)
- The module must be on GitHub and must be a private repo with at least one release tag in x.y.z format.
- GitHub repository must have HarshiCorp Licence (Version ~> 2).
- Module GitHub repositories must use this three-part name format **module-<PROVIDER>-<NAME>** where <NAME> segment can contain additional hyphens.

Explanation

Requirements

The list below contains all the requirements for publishing a module. Meeting the requirements for publishing a module is extremely easy. The list may appear long only to ensure we're detailed, but adhering to the requirements should happen naturally.

* **GitHub.** The module must be on GitHub and must be a public repo. This is only a requirement for the public registry. If you're using a private registry, you may ignore this requirement.

* **Named terraform-<PROVIDER>-<NAME>.** Module repositories must use this three-part name format, where <NAME> reflects the type of infrastructure the module manages and <PROVIDER> is the main provider where it creates that infrastructure.
The <NAME> segment can contain additional hyphens. Examples: terraform-google-vault or terraform-aws-ec2-instance.

* **Repository description.** The GitHub repository description is used to populate the short description of the module. This should be a simple one-sentence description of the module.

* **Standard module structure.** The module must adhere to the standard module structure. This allows the registry to inspect your module and generate documentation, track resource usage, parse submodules and examples, and more.

* **x.y.z tags for releases.** The registry uses tags to identify module versions. Release tag names must be a semantic version, which can optionally be prefixed with a v. For example, v1.0.4 and 0.9.2. To publish a module initially, at least one release tag must be present. Tags that don't look like version numbers are ignored.

Click here to know more about Publishing a Module

Question 44: Skipped
Peter uses configuration given below to provision an aws_db_instance and uses environment variables to provide values to variables used within the resource block secretly. Can Terraform still expose these sensitive values like passwords in the state file?

```
# ............

variable "db_instance_name" {}
variable "db_username" {}
variable "db_password" {}

resource "aws_db_instance" "quiz_experts_demo" {
allocated_storage    = 20
storage_type         = "gp2"
engine               = "mysql"
engine_version       = "5.7"
instance_class       = "db.t2.micro"
name                 = var.db_instance_name
username             = var.db_username
password             = var.db_password
}

# ............
```

- No, Terraform states are encrypted at rest by default.
- Yes, all arguments, including the username and password, will be stored in the raw state as plain-text.
  (Correct)
- Yes, all arguments will be stored in the raw state as plain-text; however, you can use sensitive argument to mark a variable containing sensitive value so that Terraform can

avoid storing it in plain-text.

- No, Terraform does not store sensitive data like username and password in the state file.

Explanation

Sensitive Data in State

Terraform state can contain sensitive data, depending on the resources in use and your definition of "sensitive." The state contains resource IDs and all resource attributes. For resources such as databases, this may contain initial passwords.

When using a local state, the state is stored in plain-text JSON files.

Recommendations

If you manage any sensitive data with Terraform (like database passwords, user passwords, or private keys), treat the state itself as sensitive data.

Storing the state remotely can provide better security.

Click here to know more about Sensitive Data in State

Question 45: Skipped
Which of the following variable block will produce an error while executing terraform validate command?

-

```
variable "amis" {
type     = map
default = []
}
```

(Correct)

-

```
variable "node_pools" {
type        = map(any)
description = "number of node pools"
}
```

-

```
variable "server_configs" {
type = object({
server_ips = list(string)
username    = string
password    = string
})
}
```

-

```
variable "cidrs" { default = [] }
```

Explanation

If both the type and default arguments are specified, the given default value must be convertible to the specified type. In the example given below -

```
variable "amis" {
  type     = map
  default = []  # This must be a valid map type instead of list.
}
```

NOTE: The keyword any maybe used to indicate that any type is acceptable.

Click here to know more about Variable Type Constraints

Question 46: Skipped
Connection blocks don't take a block label, and can be nested within either a resource or a provisioner.

-

False

-

True
(Correct)

Explanation

Connection blocks don't take a block label, and can be nested within either a resource or a provisioner.

***** A connection block nested directly within a resource affects all of that resource's provisioners.

```
resource "aws_instance" "quiz_experts" {
  ami           = "ami-04579a6a597"
  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"
  ]
 }
}
```

* A connection block nested in a provisioner block only affects that provisioner, and overrides any resource-level connection settings.

```
resource "aws_instance" "quiz_experts" {
 # Something
 # Something
# Copies the file as the root user using SSH
 provisioner "file" {
   source      = "conf/myapp.conf"
   destination = "/etc/myapp.conf"

connection {
    type     = "ssh"
    user     = "root"
    password = "${var.root_password}"
    host     = "${var.host}"
  }
 }
}
```

Use case: One use case for providing multiple connections is to have an initial provisioner connect as the root user to set up user accounts, and have subsequent provisioners connect as a user with more limited permissions.

Click here to know more about Provisioners Connection

Question 47: Skipped
You delete one or more resources from a Terraform configuration file since the configuration no longer exists. How does Terraform determine what resources it should delete and in which order?

- Terraform will no longer manage these resources. We have to delete these manually from the cloud provider management console.
- Terraform retains a copy of the most recent set of dependencies within the state. Even if you delete one or more items from the configuration, Terraform can still determine what to delete and the correct order for destruction from the state.
  (Correct)
- Since configuration no longer exists, Terraform has no way to delete such resources.
- This is still an open issue at Terraform. You should have used count = 0 parameter instead of removing resources from Terraform configuration, which could help Terraform know that it should remove these resources.

Explanation

Terraform requires some sort of database to map Terraform config to the real world. When you have a resource resource "aws_instance" "quiz_experts" in your configuration, Terraform uses this map to know that instance i-abcd1234 is represented by that resource.

Alongside the mappings between resources and remote objects, Terraform also tracks metadata such as resource dependencies and record it in state files.

Terraform typically uses the configuration to determine the dependency order. However, when you delete a resource from a Terraform configuration, Terraform must know how to delete that resource. Terraform can see that a mapping exists for a resource, not in your configuration, and plan to destroy. However, since the configuration no longer exists, the order cannot be determined from the configuration alone.

To ensure correct operation, Terraform retains a copy of the most recent set of dependencies within the state. Now Terraform can still determine the correct order for destruction from the state when you delete one or more items from the configuration.

Click here to know more about Terraform State

Question 48: Skipped
Can the expression of a local value refer to other locals?

- Yes
  (Correct)
- No

Explanation

The expression of a local value can refer to other locals, but as usual reference cycles are not allowed. That is, a local cannot refer to itself or to a variable that refers (directly or indirectly) back to it.

Example:

```
locals {
 # Common tags to be assigned to all resources
 common_tags = {
   Service = local.service_name
   Owner   = local.owner
 }
```

```
}
resource "aws_instance" "example" {
  # ...

tags = local.common_tags
}
```

Click here to know more about Locals

Question 49: Skipped
Fill in the blanks: The _____ environment variable can enable detailed Terraform logs to appear on _____.

- First - TF_CRASH_LOG
  Second - stdout
- First - TF_LOGS
  Second - stdout
- First - TF_LOG
  Second - stderr
  (Correct)
- First - TF_PATH
  Second - stderr
- First - TF_LOG
  Second - stdout

Explanation

Debugging Terraform

The TF_LOG environment variable can enable detailed Terraform 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.

Example

```
export TF_LOG=TRACE
```

Click here to know more about debugging in Terraform

Question 50: Skipped
Can another process unlock local state files?

- Yes, local state files can be unlocked by another process.
- Local state files cannot be unlocked by another process.
  (Correct)

Explanation

The behavior of this lock is dependent on the backend being used. Local state files cannot be unlocked by another process.

Click here to know about force-unlock command

Question 51: Skipped
Which of the following options can enable the most verbose logs and write those logs to a file? Choose *TWO* correct answers.

- 578. export TF_LOG="TRACE"
  579. export TF_LOG_FILE="terraform.txt"
- 580. export TF_LOG="DEBUG"
  581. export TF_LOG_PATH="terraform.txt"
- 582. export TF_LOG="RANDOM"
  583. export TF_LOG_PATH="terraform.txt"
  (Correct)
- 584. export TF_LOG="TRACE"
  585. export TF_FILE="terraform.txt"
- 586. export TF_LOG="TRACE"
  587. export TF_LOG_PATH="terraform.txt"
  (Correct)

Explanation

Terraform has detailed logs that 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.

Therefore export TF_LOG="RANDOM" or export TF_LOG="TRACE" both can set the most verbose log level.

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.

```
export TF_LOG_PATH="terraform.txt"
```

Since Terraform writes logs to *stderr* you can also use typical Linux redirection syntax to write logs into a file, but it is not recommended and you have to add it in every Terraform command.

```
terraform plan 2>> terraform.txt
terraform apply 2>> terraform.txt
```

Click here to know more about TF_LOG and TF_LOG_PATH variables

Question 52: Skipped
Which of the following variable type allows multiple values of several distinct types to be grouped together as a single value? Choose *TWO* correct answers.

- Tuple
(Correct)
- Set
- List
- Object
(Correct)
- Map

Explanation

Structural Types

A *structural* type allows multiple values of *several distinct types* to be grouped together as a single value. Structural types require a *schema* as an argument, to specify which types are allowed for which elements.

The two kinds of structural type in the Terraform language are:

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

The schema for object types is { <KEY> = <TYPE>, <KEY> = <TYPE>, ... } — a pair of curly braces containing a comma-separated series of <KEY> = <TYPE> pairs. Values that match the object type must contain *all* of the specified keys, and the value for each key must match its specified type. (Values with *additional* keys can still match an object type, but the extra attributes are discarded during type conversion.)

For Example: An object type of object({ name=string, age=number }) would match a value like the following:

```
{
  name = "John"
  age  = 52
}
```

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

The schema for tuple types is [<TYPE>, <TYPE>, ...] — a pair of square brackets containing a comma-separated series of types. Values that match the tuple type must have *exactly* the same number of elements (no more and no fewer), and the value in each position must match the specified type for that position.

For Example: A tuple type of tuple([string, number, bool]) would match a value like the following:

```
["a", 15, true]
```

Click here to know more about Tuple and Objects

Question 53: Skipped
Every Terraform configuration has at least one module.

- True
(Correct)
- False

Explanation

Every Terraform configuration has at least one module, known as its *root module*, which consists of the resources defined in the .tf files in the main working directory.

Click here to know more about Terraform Modules

Question 54: Skipped
The terraform import command currently can only import one resource at a time.

- False
- True
(Correct)

Explanation

The terraform import command currently can only import one resource at a time. This means you can't yet point Terraform import to an entire collection of resources such as an AWS VPC and import all of it. This workflow will be improved in a future version of Terraform.

Warning: Terraform expects that each remote object it is managing will be bound to only one resource address, which is normally guaranteed by Terraform itself having created all objects. If you import existing objects into Terraform, be careful to import each remote object to only one Terraform resource address. If you import the same object multiple times, Terraform may exhibit unwanted behavior. For more information on this assumption, see the State section.

Click here to know more about terraform import command

Question 55: Skipped
Your company has been using a paid Cloud Management tool to manage resources in AWS. Due to some upper management decisions, the company wants to discontinue this paid Cloud Management tool, and leverage open-source Terraform to move towards Infrastructure as Code and save cost. Now your boss asked you to explore Terraform to manage existing AWS resources. How can it be done without recreating existing resources?

- Add a provider block with valid AWS access key and secrete key to Terraform configuration, and run terraform refresh followed by terraform apply to manage existing resources with Terraform.
- You can use the terraform export command to export existing infrastructure to Terraform's state file, so that Terraform is aware of and can manage that infrastructure.
- Existing resources can not be managed by open-source Terraform. You need a paid version to import existing infrastructure directly to the Terraform Cloud state.
- You have to stop your existing resources, and then use terraform import command with resource-id to import all resources one by one to Terraform state.
- You can use the terraform import command to add existing infrastructure to Terraform's state file, so that Terraform is aware of and can manage that infrastructure.
(Correct)

Explanation

Terraform Usage

Terraform is able to import existing infrastructure. This allows you to 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.

Currently State Only

The current implementation of Terraform import can only import resources into the state. It does not generate configuration. A future version of Terraform will also generate configuration.

Because of this, prior to running terraform import it is necessary to write manually a resource configuration block for the resource, to which the imported object will be mapped.

Import Usage

To import a resource, first write a resource block for it in your configuration, establishing the name by which it will be known to Terraform:

```
resource "aws_instance" "quiz_experts_vm" {
  # ...instance configuration...
}
```

The name "quiz_experts_vm" here is local to the module where it is declared and is chosen by the configuration author. This is distinct from any ID issued by the remote system, which may change over time while the resource name remains constant.

If desired, you can leave the body of the resource block blank for now and return to fill it in once the instance is imported.

Now terraform import can be run to attach an existing instance to this resource configuration:

```
$ terraform import aws_instance.quiz_experts_vm i-c4gd1563l
```

This command locates the AWS instance with ID i-c4gd1563l. Then it attaches the existing settings of the instance, as described by the EC2 API, to the name aws_instance.quiz_experts_vm of a module. In this example, the module path implies that the root module is used. Finally, the mapping is saved in the Terraform state.

Click here to know more about terraform import command usage

Question 56: Skipped

Which of the following describes the right ways of using a local path as module sources? Choose *TWO* correct answers.

-

```
module "consul" {
source = "modules/consul"
}
```

-

```
module "consul" {
source = "../consul"
}
```

(Correct)

-

```
module "consul" {
source = "consul"
}
```

-

```
module "consul" {
source = "./consul"
}
```

(Correct)

Explanation

Local path references allow for factoring out portions of a configuration within a single source repository.

```
module "consul" {
  source = "./consul"
}
```

A local path must begin with either ./ or ../ to indicate that a local path is intended, to distinguish from a module registry address.

Local paths are special in that they are not "installed" in the same sense that other sources are: the files are already present on local disk (possibly as a result of installing a parent module) and so can just be used directly. Their source code is automatically updated if the parent module is upgraded.

Click here to know more about Terraform Module Sources

Question 57: Skipped

What command you can use to get human-readable output from a state or plan file?

- terraform plan
- terraform show
  (Correct)
- terraform state
- terraform workspace list

Explanation

The terraform show command is used to provide human-readable output from a state or plan file. This can be used to inspect a plan to ensure that the planned operations are expected, or to inspect the current state as Terraform sees it.

Click here to know more about show command

Continue

Retake test

Fullscreen

Expand

Course content

Practice tests

- Progress cannot be changed for this item
  Practice Test 1: Terraform Associate Certification
- Progress cannot be changed for this item
  Practice Test 2: Terraform Associate Certification
- Progress cannot be changed for this item
  Practice Test 3: Terraform Associate Certification
- Progress cannot be changed for this item
  Practice Test 4: Important Questions

Overview

Q&AQuestions and answers

Notes

Announcements

From <https://cognizant.udemy.com/course/hashicorp-certified-terraform-associate-2020/learn/quiz/4968266/result/403244050#overview>

Question 1: Skipped
From the answers below, select the advantages of using Infrastructure as Code. (select four)

- Safely test modifications using a "dry run" before applying any actual changes
(Correct)
- Easily change and update existing infrastructure
(Correct)
- Provide reusable modules for easy sharing and collaboration
(Correct)
- Easily integrate with application workflows (GitLab Actions, Azure DevOps, CI/CD tools)
(Correct)
- Provide a codified workflow to develop customer-facing applications

Explanation

Infrastructure as Code is **not** used to develop applications, but it can be used to help deploy or provision those applications to a public cloud provider or on-premises infrastructure.

All of the others are benefits to using Infrastructure as Code over the traditional way of managing infrastructure, regardless if it's public cloud or on-premises.

Question 2: Skipped
In order to make a Terraform configuration file dynamic and/or reusable, static values should be converted to use what?

- module
- input parameters
(Correct)
- output value
- regular expressions

Explanation

Input variables serve as parameters for a Terraform module, allowing aspects of the module to be customized without altering the module's own source code, and allowing modules to be shared between different configurations.

Question 3: Skipped
True or False: Provisioners should only be used as a last resort.

- false
- true
(Correct)

Explanation

Provisioners are used to execute scripts on a local or remote machine as part of resource creation or destruction. Provisioners can be used to bootstrap a resource, cleanup before destroy, run configuration management, etc. Even if the functionality you need is not available in a provider today, HashiCorp suggests that you consider local-exec usage as a temporary workaround and to open an issue in the relevant provider's repo to discuss adding first-class support.

Question 4: Skipped
Which of the following Terraform files should be ignored by Git when committing code to a repo? (select two)

- variables.tf
- output.tf
- terraform.tfstate
(Correct)
- terraform.tfvars
(Correct)

Explanation

The .gitignore file should be configured to ignore Terraform files that either contain sensitive data or aren't required to save.

The terraform.tfstate file contains the terraform state of a specific environment and doesn't need to be preserved in a repo. The terraform.tfvars file may contain sensitive data, such as passwords or IP addresses of an environment that you may not want to share with others.

Question 5: Skipped
What are the benefits of using Infrastructure as Code? (select five)

- Infrastructure as Code provides configuration consistency and standardization

among deployments
(Correct)

- Infrastructure as Code is relatively simple to learn and write, regardless of a user's prior experience with developing code
(Correct)

- Infrastructure as Code is easily repeatable, allowing the user to reuse code to deploy similar, yet different resources
(Correct)

- Infrastructure as Code allows a user to turn a manual task into a simple, automated deployment
(Correct)

- Infrastructure as Code gives the user the ability to recreate an application's infrastructure for disaster recovery scenarios
(Correct)

- Infrastructure as Code easily replaces development languages such as Go and .Net for application development

Explanation

If you are new to infrastructure as code as a concept, it is the process of managing infrastructure in a file or files rather than manually configuring resources in a user interface. A resource in this instance is any piece of infrastructure in a given environment, such as a virtual machine, security group, network interface, etc.

At a high level, Terraform allows operators to use HCL to author files containing definitions of their desired resources on almost any provider (AWS, GCP, GitHub, Docker, etc) and automates the creation of those resources at the time of application.

Question 6: Skipped
Using multi-cloud and provider-agnostic tools provides which of the following benefits? (select two)

- increased risk due to all infrastructure relying on a single tool for management
- can be used across major cloud providers and VM hypervisors
(Correct)
- slower provisioning speed allows the operations team to catch mistakes before they are applied
- operations teams only need to learn and manage a single tool to manage infrastructure, regardless of where the infrastructure is deployed
(Correct)

Explanation

Using a tool like Terraform can be advantageous for organizations deploying workloads across multiple public and private cloud environments. Operations teams only need to learn a single tool, single language, and can use the same tooling to enable a DevOps-like experience and workflows.

Question 7: Skipped
True or False: Workspaces provide identical functionality in the open-source, Terraform Cloud, and Enterprise versions of Terraform.

- False
(Correct)
- True

Explanation

Workspaces, managed with the `terraform workspace` command, isn't the same thing as Terraform Cloud's workspaces. Terraform Cloud workspaces act more like completely separate working directories.

CLI workspaces (OSS) are just alternate state files.

Question 8: Skipped
Which of the following is considered a Terraform plugin?

- Terraform tooling
- Terraform language
- Terraform provider
(Correct)
- Terraform logic

Explanation

Terraform is built on a plugin-based architecture. All providers and provisioners that are used in Terraform configurations are plugins, even the core types such as AWS and Heroku. Users of Terraform are able to write new plugins in order to support new functionality in Terraform.

Question 9: Skipped
After running into issues with Terraform, you need to enable verbose logging to assist with troubleshooting the error. Which of the following values provides the MOST verbose logging?

- ERROR
- WARN
-

DEBUG

- TRACE
  (Correct)
- INFO

Explanation

Terraform has detailed logs that 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.

Question 10: Skipped
When configuring a remote backend in Terraform, it might be a good idea to purposely omit some of the required arguments to ensure secrets and other relevant data are not inadvertently shared with others. What are the ways the remaining configuration can be added to Terraform so it can initialize and communicate with the backend? (select three)

- command-line key/value pairs
  (Correct)
- interactively on the command line
  (Correct)
- directly querying HashiCorp Vault for the secrets
- use the -backend-config=PATH to specify a separate config file
  (Correct)

Explanation

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*.

With a partial configuration, the remaining configuration arguments must be provided as part of the initialization process. There are several ways to supply the remaining arguments:

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.

Question 11: Skipped
Given the Terraform configuration below, in which order will the resources be created?

```
resource "aws_instance" "web_server" {
ami = "i-abdce12345"
instance_type = "t2.micro"
}

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

- no resources will be created
- aws_eip will be created first
  aws_instance will be created second
- aws_instance will be created first
  aws_eip will be created second
  (Correct)
- resources will be created simultaneously

Explanation

The aws_instance will be created first, and then aws_eip will be created second due to the aws_eip's resource dependency of the aws_instance id

More information on resource dependencies can be found at this link.

Question 12: Skipped
In regards to Terraform state file, select all the statements below which are correct: (select four)

- using the mask feature, you can instruct Terraform to mask sensitive data in the state file
- storing state remotely can provide better security
  (Correct)
- the Terraform state can contain sensitive data, therefore the state file should be protected from unauthorized access
  (Correct)

- the state file is always encrypted at rest
- when using local state, the state file is stored in plain-text
  (Correct)
- Terraform Cloud always encrypts state at rest
  (Correct)

Explanation

Terraform state can contain sensitive data, depending on the resources in use and your definition of "sensitive." The state contains resource IDs and all resource attributes. For resources such as databases, this may contain initial passwords.

When using local state, state is stored in plain-text JSON files.

If you manage any sensitive data with Terraform (like database passwords, user passwords, or private keys), treat the state itself as sensitive data.

Storing Terraform state remotely can provide better security. As of Terraform 0.9, Terraform does not persist state to the local disk when remote state is in use, and some backends can be configured to encrypt the state data at rest.

Question 13: Skipped
You want to use terraform import to start managing infrastructure that was not originally provisioned through infrastructure as code. Before you can import the resource's current state, what must you do in order to prepare to manage these resources using Terraform?

- update the configuration file to include the new resources
  (Correct)
- run terraform refresh to ensure that the state file has the latest information for existing resources.
- shut down or stop using the resources being imported so no changes are inadvertently missed
- modify the Terraform state file to add the new resources

Explanation

The current implementation of Terraform import can only import resources into the state. It does not generate a configuration. Because of this, and prior to running terraform import, it is necessary to manually write a resource configuration block for the resource to which the imported object will be mapped.

First, add the resources to the configuration file:

```
resource "aws_instance" "example" {
  # ...instance configuration...
}
```

Then run the following command:

```
$ terraform import aws_instance.example i-abcd1234
```

Question 14: Skipped
After executing a terraform apply, you notice that a resource has a tilde (~) next to it. What does this infer?

- the resource will be created
- the resource will be destroyed and recreated
- Terraform can't determine how to proceed due to a problem with the state file
- the resource will be updated in place
  (Correct)

Explanation

The prefix -/+ means that Terraform will destroy and recreate the resource, rather than updating it in-place. Some attributes and resources can be updated in-place and are shown with the ~ prefix.

```
  ~ root_block_device {
      ~ delete_on_termination = true -> (known after apply)
      ~ iops                  = 100 -> (known after apply)
      ~ volume_id             = "vol-0079e485d9e28a8e5" -> (known after apply)
      ~ volume_size           = 8 -> (known after apply)
      ~ volume_type           = "gp2" -> (known after apply)
    }
}
```

Question 15: Skipped
Which of the following actions are performed during a terraform init? (select three)

- provisions the declared resources in your configuration
- initializes downloaded and/or installed providers
  (Correct)
- download the declared providers which are supported by HashiCorp
  (Correct)
- initializes the backend configuration
  (Correct)

Explanation

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.

Question 16: Skipped
Which flag would be used within a Terraform configuration block to identify the specific version of a provider required?

- required_versions
- required-version
- required_providers
  (Correct)
- required-provider

Explanation

For production use, you should constrain the acceptable provider versions via configuration file to ensure that new versions with breaking changes will not be automatically installed by terraform init in the future. When terraform init is run *without* provider version constraints, it prints a suggested version constraint string for each provider

For example:

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

Question 17: Skipped
True or False: State is a requirement for Terraform to function.

- False
- True
  (Correct)

Explanation

Terraform requires some sort of database to map Terraform config to the real world. When you have a resource in your configuration, Terraform uses this map to know how that resource is represented. Therefore, to map configuration to resources in the real world, Terraform uses its own state structure.

Question 18: Skipped
What is a downside to using a Terraform provider, such as the Vault provider, to interact with sensitive data, such as reading secrets from Vault?

- Terraform and Vault must be running on the same version
- Terraform and Vault must be running on the same physical host
- secrets are persisted to the state file and plans
  (Correct)
- Terraform requires a unique auth method to work with Vault

Explanation

Interacting with Vault from Terraform causes any secrets that you read and write to be persisted in both Terraform's state file *and* in any generated plan files. For any Terraform module that reads or writes Vault secrets, these files should be treated as sensitive and protected accordingly.

Question 19: Skipped
What happens when a terraform apply command is executed?

- creates the execution plan for the deployment of resources
- reconciles the state Terraform knows about with the real-world infrastructure
- the backend is initialized and the working directory is prepped
- applies the changes required in the target infrastructure in order to reach the desired configuration
  (Correct)

Explanation

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.

Question 20: Skipped
Which of the following represents a feature of Terraform Cloud that is NOT free to customers?

- private module registry
- workspace management
- roles and team management
  (Correct)
- VCS integration

Explanation

Information on the comparisons of Terraform Cloud can be found at this link.

Question 21: Skipped
Which of the following best describes the default local backend?

- The local backend is where Terraform Enterprise stores logs to be processed by an log collector
- The local backend is the directory where resources deployed by Terraform have direct access to in order to update their current state
- The local backend stores state on the local filesystem, locks the state using system APIs, and performs operations locally.
(Correct)
- The local backend is how Terraform connects to public cloud services, such as AWS, Azure, or GCP.

Explanation

Information on the default local backend can be found at this link.

Example:

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

Question 22: Skipped
By default, where does Terraform store its state file?

- current working directory
(Correct)
- remotely using Terraform Cloud
- shared directory
- Amazon S3 bucket

Explanation

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

Question 23: Skipped
What Terraform feature is shown in the example below?

```
resource "aws_security_group" "example" {
name = "sg-app-web-01"

dynamic "ingress" {
for_each = var.service_ports
content {
from_port = ingress.value
to_port   = ingress.value
protocol  = "tcp"
}
}
}
```

- data source
- dynamic block
(Correct)
- conditional expression
- local values

Explanation

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

Question 24: Skipped
Which of the following connection types are supported by the remote-exec provisioner? (select two)

- ssh
(Correct)
- rdp
- winrm
(Correct)
- smb

Explanation

The remote-exec provisioner invokes a script on a remote resource after it is created. The remote-exec provisioner supports both ssh and winrm type connections.

More information on remote-exec can be found at this link.

Question 25: Skipped
Select the answer below that completes the following statement:

Terraform Cloud can be managed from the CLI but requires _____?

- a TOTP token
-

authentication using MFA

- an API token
(Correct)
- a username and password

Explanation

API and CLI access are managed with API tokens, which can be generated in the Terraform Cloud UI. Each user can generate any number of personal API tokens, which allow access with their own identity and permissions. Organizations and teams can also generate tokens for automating tasks that aren't tied to an individual user.

Question 26: Skipped
Why is it a good idea to declare the required version of a provider in a Terraform configuration file?

Larger image

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

- to match the version number of your application being deployed via Terraform
- to remove older versions of the provider
- providers are released on a separate schedule from Terraform itself; therefore a newer version could introduce breaking changes
(Correct)
- to ensure that the provider version matches the version of Terraform you are using

Explanation

Providers are plugins released on a separate rhythm from Terraform itself, and so they have their own version numbers. For production use, you should constrain the acceptable provider version via configuration. This helps to ensure that new versions with potentially breaking changes will not be automatically installed by terraform init in the future.

Question 27: Skipped
You have been given requirements to create a security group for a new application. Since your organization standardizes on Terraform, you want to add this new security group with the fewest number of lines of code. What feature could you use to iterate over a list of required tcp ports to add to the new security group?

- terraform import
- dynamic block
(Correct)
- splat expression
- dynamic backend

Explanation

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.

You can find more information on dynamic blocks using this link.

Question 28: Skipped
What happens when a terraform plan is executed?

- reconciles the state Terraform knows about with the real-world infrastructure
- creates an execution plan and determines what changes are required to achieve the desired state in the configuration files.
(Correct)
- applies the changes required in the target infrastructure in order to reach the desired configuration
- the backend is initialized and the working directory is prepped

Explanation

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.

After a plan has been run, it can be executed by running a terraform apply

Question 29: Skipped
Select the feature below that best completes the sentence:

The following list represents the different types of _____ available in Terraform.

621. **max**
622. **min**
623. **join**
624. **replace**
625. **list**
626. **length**
627. **range**

- functions
  (Correct)
- data sources
- backends
- named values

Explanation

The Terraform language includes a number of built-in functions that you can call from within expressions to transform and combine values. The Terraform language does not support user-defined functions, and only the functions built into the language are available for use.

Question 30: Skipped
What is the best and easiest way for Terraform to read and write secrets from HashiCorp Vault?

- integration with a tool like Jenkins
- CLI access from the same machine running Terraform
- API access using the AppRole auth method
- Vault provider
  (Correct)

Explanation

The Vault provider allows Terraform to read from, write to, and configure Hashicorp Vault.

Question 31: Skipped
When using parent/child modules to deploy infrastructure, how would you export a value from one module to import into another module.

For example, a module dynamically deploys an application instance or virtual machine, and you need the IP address in another module to configure a related DNS record in order to reach the newly deployed application.

- configure an output value in the application module in order to use that value for the DNS module
  (Correct)
- export the value using terraform export and input the value using terraform input
- configure the pertinent provider's configuration with a list of possible IP addresses to use
- preconfigure the IP address as a parameter in the DNS module

Explanation

Output values are like the return values of a Terraform module and have several uses such as a child module using those outputs to expose a subset of its resource attributes to a parent module.

Question 32: Skipped
What Terraform command can be used to inspect the current state file?

Example:

Larger image

```
# aws_instance.example:
resource "aws_instance" "example" {
    ami                          = "ami-2757f631"
    arn                          = "arn:aws:ec2:us-east-1:130490850807:instance/i-0
    associate_public_ip_address  = true
    availability_zone            = "us-east-1c"
    cpu_core_count               = 1
    cpu_threads_per_core         = 1
    disable_api_termination      = false
    ebs_optimized                = false
    get_password_data            = false
    id                           = "i-0bbf06244e44211d1"
    instance_state               = "running"
    instance_type                = "t2.micro"
```

- terraform state
- terraform read
- terraform show
  (Correct)
- terraform inspect

Explanation

The terraform show command is used to provide human-readable output from a state or plan file. This can be used to inspect a plan to ensure that the planned operations are expected, or to inspect the current state as Terraform sees it.

Machine-readable output can be generated by adding the -json command-line

flag.

Note: When using the -json command-line flag, any sensitive values in Terraform state will be displayed in plain text.

Question 33: Skipped
True or False: You can migrate the Terraform backend but only if there are no resources currently being managed.

- False
  (Correct)
- True

Explanation

If you are already using Terraform to manage infrastructure, you probably want to transfer to another backend, such as Terraform Cloud, so you can continue managing it. By migrating your Terraform state, you can hand off infrastructure without de-provisioning anything.

Question 34: Skipped
Select two answers to complete the following sentence:

Before a new provider can be used, it must be _____ and _____.

- uploaded to source control
- initialized
  (Correct)
- declared in the configuration
  (Correct)
- approved by HashiCorp

Explanation

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.

Question 35: Skipped
In the example below, where is the value of the DNS record's IP address originating from?

```
resource "aws_route53_record" "www" {
zone_id = aws_route53_zone.primary.zone_id
name    = "www.helloworld.com"
type    = "A"
ttl     = "300"
records = [module.web_server.instance_ip_addr]
}
```

- the output of a module named web_server
  (Correct)
- the regular expression named module.web_server
- value of the web_server parameter from the variables.tf file
- by querying the AWS EC2 API to retrieve the IP address

Explanation

In a parent module, outputs of child modules are available in expressions as module.<MODULE NAME>.<OUTPUT NAME>. For example, if a child module named web_server declared an output named instance_ip_addr, you could access that value as module.web_server.instance_ip_addr.

Question 36: Skipped
Your organization has moved to AWS and has manually deployed infrastructure using the console. Recently, a decision has been made to standardize on Terraform for all deployments moving forward.

What can you do to ensure that all existing is managed by Terraform moving forward without interruption to existing services?

- using terraform import, import the existing infrastructure into your Terraform state
  (Correct)
- submit a ticket to AWS and ask them to export the state of all existing resources and use terraform import to import them into the state file
- resources that are manually deployed in the AWS console cannot be imported by Terraform
- delete the existing resources and recreate them using new a Terraform configuration so Terraform can manage them moving forward

Explanation

Terraform is able to import existing infrastructure. This allows you to 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 sure you're confident that you can use Terraform in the future if it currently doesn't support every AWS service or feature you need today.

Question 37: Skipped
True or False: Multiple providers can be declared within a single Terraform configuration file.

- false
- true
  (Correct)

Explanation

Multiple provider blocks can exist if a Terraform configuration is composed of multiple providers, which is a common situation. To add multiple providers in your configuration, declare the providers, and create resources associated with those providers.

Question 38: Skipped
In Terraform Enterprise, a workspace can be mapped to how many VCS repos?

- 1
  (Correct)
- 5
- 3
- 2

Explanation

A workspace can only be configured to a single VCS repo, however, multiple workspaces can use the same repo, if needed. A good explanation of how to configure your code repositories can be found here.

Question 39: Skipped
What are some of the features of Terraform state? (select three)

- mapping configuration to real-world resources
  (Correct)
- inspection of cloud resources
- increased performance
  (Correct)
- determining the correct order to destroy resources
  (Correct)

Explanation

See this page on the purpose of Terraform state and the benefits it provides.

Question 40: Skipped
Which of the following allows Terraform users to apply policy as code to enforce standardized configurations for resources being deployed via infrastructure as code?

- workspaces
- functions
- module registry
- sentinel
  (Correct)

Explanation

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.

Question 41: Skipped
In regards to deploying resources in multi-cloud environments, what are some of the benefits of using Terraform rather than a provider's native tooling? (select three)

- Terraform can help businesses deploy applications on multiple clouds and on-premises infrastructure
  (Correct)
- Terraform simplifies management and orchestration, helping operators build large-scale, multi-cloud infrastructure
  (Correct)
- Terraform can manage cross-cloud dependencies
  (Correct)
- Terraform is not cloud-agnostic and can be used to deploy resources across a single public cloud

Explanation

Terraform is a cloud-agnostic tool, and therefore isn't limited to a single cloud provider, such as AWS CloudFormation or Azure Resource Manager. Terraform supports all of the major cloud providers and allows IT organizations to focus on learning a single tool for deploying its infrastructure, regardless of what platform it's being deployed on

Question 42: Skipped
What are some of the problems of how infrastructure was traditionally managed before Infrastructure as Code? (select three)

- Pointing and clicking in a management console is a scalable approach and

reduces human error as businesses are moving to a multi-cloud deployment model

- Traditional deployment methods are not able to meet the demands of the modern business where resources tend to live days to weeks, rather than months to years
(Correct)
- Traditionally managed infrastructure can't keep up with cyclic or elastic applications
(Correct)
- Requests for infrastructure or hardware required a ticket, increasing the time required to deploy applications
(Correct)

Explanation

Businesses are making a transition where traditionally-managed infrastructure can no longer meet the demands of today's businesses. IT organizations are quickly adopting the public cloud, which is predominantly API-driven.

To meet customer demands and save costs, application teams are architecting their applications to support a much higher level of elasticity, supporting technology like containers and public cloud resources. These resources may only live for a matter of hours; therefore the traditional method of raising a ticket to request resources is no longer a viable option

Pointing and clicking in a management console is NOT scale and increases the change of human error.

Question 43: Skipped
Which Terraform command will force a marked resource to be destroyed and recreated on the next apply?

- terraform destroy
- terraform refresh
- terraform fmt
- terraform taint
(Correct)

Explanation

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. The next terraform apply will implement this change.

Question 44: Skipped
What does the command terraform fmt do?

- deletes the existing configuration file
- updates the font of the configuration file to the official font supported by HashiCorp
- formats the state file in order to ensure the latest state of resources can be obtained
- rewrite Terraform configuration files to a canonical format and style
(Correct)

Explanation

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.

Other Terraform commands that generate Terraform configuration will produce configuration files that conform to the style imposed by terraform fmt, so using this style in your own files will ensure consistency.

Question 45: Skipped
HashiCorp offers multiple versions of Terraform, including Terraform open-source, Terraform Cloud, and Terraform Enterprise. Which of the following Terraform features are only available in the Enterprise edition? (select four)

- Clustering
(Correct)
- Private Network Connectivity
(Correct)
- Audit Logs
(Correct)
- Private Module Registry
- SAML/SSO
(Correct)
- Sentinel

Explanation

While there are a ton of features that are available to open source users, many features that are part of the Enterprise offering are geared towards larger teams and enterprise functionality. To see what specific features are part of Terraform Cloud and Terraform Enterprise, check out this link.

Question 46: Skipped
What is the purpose of using the local-exec provisioner? (select two)

- executes a command on the resource to invoke an update to the Terraform state
- to invoke a local executable
(Correct)
- ensures that the resource is only executed in the local infrastructure where Terraform is deployed
- to execute one or more commands on the machine running Terraform
(Correct)

Explanation

The local-exec provisioner invokes a local executable after a resource is created. This invokes a process on the machine running Terraform, not on the resource.

Note that even though the resource will be fully created when the provisioner is run, there is no guarantee that it will be in an operable state - for example, system services such as sshd may not be started yet on compute resources.

Question 47: Skipped
Which of the following best describes a Terraform provider?

- serves as a parameter for a Terraform module that allows a module to be customized
- a plugin that Terraform uses to translate the API interactions with the service or provider
(Correct)
- a container for multiple resources that are used together
- describes an infrastructure object, such as a virtual network, compute instance, or other components

Explanation

A provider is responsible for understanding API interactions and exposing resources. Providers generally are an IaaS (e.g., Alibaba Cloud, AWS, GCP, Microsoft Azure, OpenStack), PaaS (e.g., Heroku), or SaaS services (e.g., Terraform Cloud, DNSimple, CloudFlare).

Question 48: Skipped
Terraform-specific settings and behaviors are declared in which configuration block type?

- resource
- terraform
(Correct)
- provider
- data

Explanation

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.

Question 49: Skipped
When multiple engineers start deploying infrastructure using the same state file, what is a feature of remote state storage that is critical to ensure the state does not become corrupt?

- state locking
(Correct)
- workspaces
- encryption
- object storage

Explanation

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

Question 50: Skipped
True or False: A list(...) may contain a number of values of the same type while an object(...) can contain a number of values of different types.

- True
(Correct)
- False

Explanation

A *collection* type allows multiple values of *one* other type to be grouped together as a single value. This includes list, map, and set.

A *structural* type allows multiple values of *several distinct types* to be grouped together as a single value. This includes object and tuple.

From <https://cognizant.udemy.com/course/terraform-associate-practice-exam/learn/quiz/4775582/result/403194250#overview>

Question 1: Skipped
During a terraform apply, a resource is successfully created but eventually fails during provisioning. What happens to the resource?

- the terraform plan is rolled back and all provisioned resources are removed
- the resource is marked as tainted
  (Correct)
- Terraform attempts to provision the resource up to three times before exiting with an error
- it is automatically deleted

Explanation

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.

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.

Question 2: Skipped
Provider dependencies are created in several different ways. Select the valid provider dependencies from the following list: (select three)

- Existence of any provider plugins found locally in the working directory.
- Use of any resource belonging to a particular provider in a resource or data block in the configuration.
  (Correct)
- Existence of any resource instance belonging to a particular provider in the current *state*.
  (Correct)
- Explicit use of a provider block in configuration, optionally including a version constraint.
  (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://www.terraform.io/docs/commands/providers.html

Question 3: Skipped
While Terraform is generally written using the HashiCorp Configuration Language (HCL), what other syntax can Terraform be expressed in?

- XML
- TypeScript
- YAML
- JSON
  (Correct)

Explanation

The constructs in the Terraform language can also be expressed in JSON syntax, which is harder for humans to read and edit but easier to generate and parse programmatically.

Question 4: Skipped
When Terraform needs to be installed in a location where it does not have internet access to download the installer and upgrades, the installation is generally known as to be _____.

- non-traditional
- a private install

disconnected

- air-gapped
  (Correct)

Explanation

A Terraform Enterprise install that is provisioned on a network that does not have Internet access is generally known as an air-gapped install. These types of installs require you to pull updates, providers, etc. from external sources vs. being able to download them directly.

Question 5: Skipped
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]
}
```

- The EC2 instance labeled web_server
  (Correct)
- The AMI used for the EC2 instance
- The S3 bucket labeled company_data
- The EIP with an id of ami-2757f631

Explanation

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.example is an explicit dependency.

Question 6: Skipped
In order to reduce the time it takes to provision resources, Terraform uses parallelism. By default, how many resources will Terraform provision concurrently?

- 10
  (Correct)
- 5
- 50
- 20

Explanation

Terraform can limit the number of concurrent operations as Terraform walks the graph using the -parallelism=n argument. The default value for this setting is 10. This setting might be helpful if you're running into API rate limits.

Question 7: Skipped
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)

- GitHub
  (Correct)
- GitHub Enterprise
  (Correct)
- Azure DevOps Server
  (Correct)
- CVS Version Control
- Bitbucket Cloud
  (Correct)

Explanation

Terraform Cloud supports the following VCS providers:

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

https://www.terraform.io/docs/cloud/vcs/index.html#supported-vcs-providers

Question 8: Skipped
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]
}
```

- internal dependency
- implicit dependency
- non-dependency resource
- explicit dependency
  (Correct)

Explanation

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.

Question 9: Skipped
Which of the following variable declarations is going to result in an error?

-
```
variable "example" {
description = "This is a test"
type        = map
default     = {"one" = 1, "two" = 2, "Three" = "3"}
}
```
-
```
variable "example" {
type = object({})
}
```
-
```
variable "example" {
  description = "This is a variable description"
  type        = list(string)
default     = {}
}
```
  (Correct)
-
```
variable "example" {}
```

Explanation

Lists are defined with [ ], maps are defined with { }.

https://www.terraform.io/docs/configuration/types.html#structural-types

Question 10: Skipped
Select the most accurate statement to describe the Terraform language from the following list.

- Terraform is a mutable, declarative, Infrastructure as Code configuration management language based on Hashicorp Configuration Language, or optionally JSON.
- Terraform is an immutable, declarative, Infrastructure as Code provisioning language based on Hashicorp Configuration Language, or optionally JSON.
  (Correct)
- Terraform is a mutable, procedural, Infrastructure as Code provisioning language based on Hashicorp Configuration Language, or optionally YAML.
- Terraform is an immutable, procedural, Infrastructure as Code configuration management language based on Hashicorp Configuration Language, or optionally JSON.

Explanation

Terraform is not a configuration management tool - https://www.terraform.io/intro/vs/chef-puppet.html

Terraform is a declarative language - https://www.terraform.io/docs/configuration/index.html

Terraform supports a syntax that is JSON compatible - https://www.terraform.io/docs/configuration/syntax-json.html

Terraform is primarily designed on immutable infrastructure principles - https://www.hashicorp.com/resources/what-is-mutable-vs-immutable-infrastructure

Question 11: Skipped
Environment variables can be used to set variables. The environment variables must be in the format "____"_*<variablename>*. Select the correct prefix string from the following list.

- TF_ENV
- TF_ENV_VAR
- TF_VAR
  (Correct)
- TF_VAR_NAME

Explanation

Environment variables can be used to set variables. The environment variables must be in the format TF_VAR_name and this will be checked last for a value. For example:

```
export TF_VAR_region=us-west-1
export TF_VAR_ami=ami-049d8641
export TF_VAR_alist='[1,2,3]'
export TF_VAR_amap='{ foo = "bar", baz = "qux" }'
```

https://www.terraform.io/docs/commands/environment-variables.html

Question 12: Skipped
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?

- terraform workspace -switch dev
- terraform workspace select dev
  (Correct)
- terraform workspace switch dev
- terraform workspace dev

Explanation

The terraform workspace select command is used to choose a different workspace to use for further operations.
https://www.terraform.io/docs/commands/workspace/select.html

Question 13: Skipped
Terraform has detailed logs which can be enabled by setting the _____ environmental variable.

- TF_DEBUG
- TF_LOG
  (Correct)
- TF_INFO
- TF_TRACE

Explanation

Terraform has detailed logs that 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.

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

Question 14: Skipped
Which of the following statements best describes the Terraform list(...) type?

- a sequence of values identified by consecutive whole numbers starting with zero.
  (Correct)
- a collection of named attributes that each have their own type.
- a collection of unique values that do not have any secondary identifiers or ordering.
- a collection of values where each is identified by a string label.

Explanation

A terraform list is a sequence of values identified by consecutive whole numbers starting with zero.

https://www.terraform.io/docs/configuration/types.html#structural-types

Question 15: Skipped
The Terraform language supports a number of different syntaxes for comments. Select all that are supported. (select three)

- /* and */
  (Correct)
- //
  (Correct)
- <* and *>
- #
  (Correct)

Explanation

Terraform supports the #, //, and /*..*/ for commenting Terraform configuration files. Please use them when writing Terraform so both you and others who are using your code have a full understanding of what the code is intended to do.

https://www.terraform.io/docs/configuration/syntax.html#comments

Question 16: Skipped
What is the result of the following terraform function call?

```
> lookup({a="hello", b="goodbye"}, "c", "what?")
```

- goodbye
- hello
- c
- what?
  (Correct)

Explanation

lookup retrieves the value of a single element from a map, given its key. If the given key does not exist, a the given default value is returned instead. In this case, the function call is searching for the key "c". Because there there is no key "c", the default vault "what?" is returned.

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

Question 17: Skipped
A user has created three workspaces using the command line - prod, dev, and test. The user wants to create a fourth workspace named stage. Which command will the user execute to accomplish this?

- terraform workspace create stage
- terraform workspace -create stage
- terraform workspace -new stage
- terraform workspace new stage
  (Correct)

Explanation

The terraform workspace new command is used to create a new workspace.
https://www.terraform.io/docs/commands/workspace/new.html

Question 18: Skipped
When using providers that require the retrieval of data, such as the HashiCorp Vault provider, in what phase does Terraform actually retrieve the data required?

- terraform apply
- terraform delete
- terraform init
- terraform plan
  (Correct)

Explanation

It is important to consider that Terraform reads from data sources during the plan phase and writes the result into the plan. For something like a Vault token which has an explicit TTL, the apply must be run before the data, or token, in this case, expires, otherwise, Terraform will fail during the apply phase.

Question 19: Skipped
A user has created a module called "*my_test_module*" and committed it to GitHub. Over time, several commits have been made with updates to the module, each tagged in GitHub with an incremental version number. Which of the following lines would be required in a module configuration block in terraform to select tagged version v1.0.4?

- source = "git::https://example.com/my_test_module.git@tag=v1.0.4"
- source = "git::https://example.com/my_test_module.git#tag=v1.0.4"
- source = "git::https://example.com/my_test_module.git?ref=v1.0.4"
  (Correct)
- source = "git::https://example.com/my_test_module.git&ref=v1.0.4"

Explanation

By default, Terraform will clone and use the default branch (referenced by HEAD) in the selected repository. You can override this using the ref argument:

```
module "vpc" {
  source = "git::https://example.com/vpc.git?ref=v1.2.0"
}
```

The value of the ref argument can be any reference that would be accepted by the git checkout command, including branch and tag names.

https://www.terraform.io/docs/modules/sources.html#selecting-a-revision

Question 20: Skipped
What feature of Terraform Cloud and/or Terraform Enterprise can you publish and maintain a set of custom modules which can be used within your organization?

- remote runs
- custom VCS integration
- private module registry
  (Correct)
- 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.

Question 21: Skipped
In the following code snippet, the block type is identified by which string?

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

- instance_type
- resource
  (Correct)
- "aws_instance"
- "db"

Explanation

The format of resource block configurations is as follows:

<block type> "<resource type>" "<local name/label>"

Question 22: Skipped
Select all features which are exclusive to Terraform Enterprise. (select three)

- Clustering
  (Correct)
- Sentinel
- Cost Estimation
- Audit Logs
  (Correct)
- SAML/SSO
  (Correct)

Explanation

Sentinel and Cost Estimation are both available in Terraform Cloud, though not at the free tier level.

https://www.hashicorp.com/products/terraform/pricing/

Question 23: Skipped
Choose the correct answer which fixes the syntax of the following Terraform code:

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

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

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

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

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

Explanation

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

Question 24: Skipped
True or False. The terraform refresh command is used to reconcile the state Terraform knows about (via its state file) with the real-world infrastructure. If drift is detected between the real-world infrastructure and the last known-state, it will modify the infrastructure to correct the drift.

- True

- False
  (Correct)

Explanation

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.

This 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.

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

Question 25: Skipped
A user runs terraform init on their RHEL based server and per the output, two provider plugins are downloaded:

```
$ terraform init

Initializing the backend...

Initializing provider plugins...
- Checking for available provider plugins...
- Downloading plugin for provider "aws" (hashicorp/aws) 2.44.0...
- Downloading plugin for provider "random" (hashicorp/random) 2.2.1...

Terraform has been successfully initialized!
```

Where are these plugins downloaded to?

- The .terraform/plugins directory in the directory terraform init was executed in.
  (Correct)
- The .terraform.plugins directory in the directory terraform init was executed in.
- The .terraform.d directory in the directory terraform init was executed in.
- /etc/terraform/plugins

Explanation

By default, terraform init downloads plugins into a subdirectory of the working directory, .terraform/plugins, so that each working directory is self-contained.

Question 26: Skipped
True or False? By default, Terraform destroy will prompt for confirmation before proceeding.

- False
- True
  (Correct)

Explanation

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:
```

Question 27: Skipped
True or False? Each Terraform workspace uses its own state file to manage the infrastructure associated with that particular workspace.

- True
  (Correct)
- False

Explanation

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.

Question 28: Skipped
When writing Terraform code, HashiCorp recommends that you use how many spaces between each nesting level?

- 1
- 2
  (Correct)
- 5
- 4

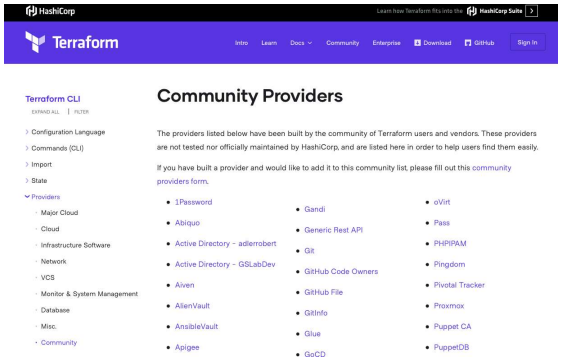
Explanation

HashiCorp style conventions state that you should use 2 spaces between each nesting level to improve the readability of Terraform configurations.

Check this link for more information

Question 29: Skipped
True or False? terraform init cannot automatically download community providers.

Larger image



- True
- False
  (Correct)

Explanation

With Terraform 0.13, terraform init can now automatically download community providers.

https://www.hashicorp.com/blog/automatic-installation-of-third-party-providers-with-terraform-0-13

Question 30: Skipped
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"
}
}
```

- 1.3.1
- 1.2.3
  (Correct)
- 1.3.0
- 1.2.9
  (Correct)

Explanation

~> 1.2.0 will match any non-beta version of the provider between >= 1.2.0 and < 1.3.0. For example, 1.2.X

https://www.terraform.io/docs/configuration/modules.html#gt-1-2-0-1

Question 31: Skipped
The following is a snippet from a Terraform configuration file:

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

provider "aws" {
region = "us-west-1"
}
```

which, when validated, results in the following error:-

```
Error: Duplicate provider configuration

on main.tf line 5:
5: provider "aws" {

A default provider configuration for "aws" was already given at
main.tf:1,1-15. If multiple configurations are required, set the "______"
argument for alternative configurations.
```

Fill in the blank in the error message with the correct string from the list below.

- label
- version
- alias
  (Correct)
- multi

Explanation

An alias meta-arguement is used when using the same provider with different configurations for different resources.

https://www.terraform.io/docs/configuration/providers.html#alias-multiple-provider-instances

Question 32: Skipped
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?

- github
  (Correct)
- s3
- terraform enterprise
- consul
- artifactory

Explanation

github is not a supported backend type.
https://www.terraform.io/docs/backends/types/index.html

Question 33: Skipped
Select the operating systems which are supported for a clustered Terraform Enterprise: (select four)

- Amazon Linux
  (Correct)
- Red Hat
  (Correct)
- Unix
- CentOS
  (Correct)
- Ubuntu
  (Correct)

Explanation

Note: (5/27/20) This question has been recently updated to reflect documentation updates on the HashiCorp website. It seems they have removed the clustering-specific requirements and are now following the standard Enterprise operating system requirements.

Terraform Enterprise currently supports running under the following operating systems for a Clustered deployment:

- Ubuntu 16.04.3 - 16.04.5 / 18.04

- Red Hat Enterprise Linux 7.4 through 7.7

- CentOS 7.4 - 7.7

- Amazon Linux

- Oracle Linux

Clusters currently don't support other Linux variants.

https://www.terraform.io/docs/enterprise/before-installing/index.html#operating-system-requirements

Question 34: Skipped
In terraform, most resource dependencies are handled automatically. Which of the following statements describes best how terraform resource dependencies are handled?

- The terraform binary contains a built-in reference map of all defined Terraform resource dependencies. Updates to this dependency map are reflected in terraform versions. To ensure you are working with the latest resource dependency map you much be running the latest version of Terraform.
- Resource dependencies are handled automatically by the depends_on meta_argument, which is set to true by default.
- Resource dependencies are identified and maintained in a file called resource.dependencies. Each terraform provider is required to maintain a list of all resource dependencies for the provider and it's included with the plugin during initialization when terraform init is executed. The file is located in the terraform.d folder.
- Terraform analyses any expressions within a resource block to find references to other objects and treats those references as implicit ordering requirements when creating, updating, or destroying resources.
  (Correct)

Explanation

Terraform analyses any expressions within a resource block to find references to other objects and treats those references as implicit ordering requirements when creating, updating, or destroying resources.

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

Question 35: Skipped
Anyone can publish and share modules on the Terraform Public Module Registry, and meeting the requirements for publishing a module is extremely easy. Select from the following list all valid requirements. (select three)

- Module repositories must use this three-part name format, *terraform-<PROVIDER>-<NAME>*.
  (Correct)
- The module must be on GitHub and must be a public repo.

(Correct)

- The module must be PCI/HIPPA compliant.
- The registry uses tags to identify module versions. Release tag names must be for the format *x.y.z*, and can optionally be prefixed with a v .
(Correct)

Explanation

The list below contains all the requirements for publishing a module. Meeting the requirements for publishing a module is extremely easy. The list may appear long only to ensure we're detailed, but adhering to the requirements should happen naturally.

GitHub. The module must be on GitHub and must be a public repo. This is only a requirement for the public registry. If you're using a private registry, you may ignore this requirement.

Named terraform-<PROVIDER>-<NAME>. Module repositories must use this three-part name format, where <NAME> reflects the type of infrastructure the module manages and <PROVIDER> is the main provider where it creates that infrastructure. The <NAME> segment can contain additional hyphens. Examples: terraform-google-vault or terraform-aws-ec2-instance.

Repository description. The GitHub repository description is used to populate the short description of the module. This should be a simple one-sentence description of the module.

Standard module structure. The module must adhere to the standard module structure. This allows the registry to inspect your module and generate documentation, track resource usage, parse submodules and examples, and more.

x.y.z tags for releases. The registry uses tags to identify module versions. Release tag names must be a semantic version, which can optionally be prefixed with a v. For example, v1.0.4 and 0.9.2. To publish a module initially, at least one release tag must be present. Tags that don't look like version numbers are ignored.

https://www.terraform.io/docs/registry/modules/publish.html#requirements

Question 36: Skipped
Which statements best describes what the local variable assignment is doing in the following code snippet:

```
variable "subnet_details" {
type = list(object({
cidr           = string
subnet_name      = string
route_table_name = string
aznum           = number
}))
}

locals {
route_tables_all = distinct([for s in var.subnet_details : s.route_table_name ])
}
```

- Create a map of route table names from a list of subnet names
- Create a list of route table names eliminating duplicates
(Correct)
- Create a distinct list of route table name objects
- Create a map of route table names to subnet names

Explanation

route_tables_all is assigned a list of unique route table names filtered from a list of objects describing subnet details, one of those object attributes being route_table_name.

Question 37: Skipped
Complete the following sentence:

For local state, the workspaces are stored directly in a...

- directory called terraform.workspaces.tfstate
- a file called terraform.tfstate
- a file called terraform.tfstate.backup
- directory called terraform.tfstate.d
(Correct)

Explanation

For local state, Terraform stores the workspace states in a directory called terraform.tfstate.d.
https://www.terraform.io/docs/state/workspaces.html#workspace-internals

Question 38: Skipped
Terraform Enterprise (also referred to as pTFE) requires what type of backend database for a clustered deployment?

- Cassandra
- MSSQL
- MySQL

- PostgreSQL
  (Correct)

Explanation

External Services mode stores the majority of the stateful data used by the instance in an external PostgreSQL database and an external S3-compatible endpoint or Azure blob storage. There is still critical data stored on the instance that must be managed with snapshots. Be sure to check the PostgreSQL Requirements for information that needs to be present for Terraform Enterprise to work. This option is best for users with expertise managing PostgreSQL or users that have access to managed PostgreSQL offerings like AWS RDS.

Question 39: Skipped
Which of the following is an *invalid* variable name?

- var1
- instance_name
- count
  (Correct)
- web

Explanation

count is a reserved word. The count parameter on resources can simplify configurations and let you scale resources by simply incrementing a number.

https://www.terraform.io/intro/examples/count.html

Question 40: Skipped
Complete the following sentence:

The terraform state command can be used to ____

- There is no such command
- view the entire state file
- modify the current state, such as removing items
  (Correct)
- refresh the existing state

Explanation

The terraform state command is used for advanced state management. 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

Question 41: Skipped
What is the result of the following terraform function call?

> index(["a", "b", "c"], "c")

- 1
- 2
  (Correct)
- true
- 0

Explanation

index finds the element index for a given value in a list starting with index 0.

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

Question 42: Skipped
Why might a user opt to include the following snippet in their configuration file?

```
terraform {
required_version = ">= 0.12"
}
```

- Terraform 0.12 introduced substantial changes to the syntax used to write Terraform configuration
  (Correct)
- versions before Terraform 0.12 were not approved by HashiCorp to be used in production
- this ensures that all Terraform providers are above a certain version to match the application being deployed
- The user wants to ensure that the application being deployed is a minimum version of 0.12

Explanation

You can use required_version to ensure that a user deploying infrastructure is using Terraform 0.12 or greater, due to the vast number of changes that were introduced. As a result, many previously written configurations had to be converted or rewritten.

Question 43: Skipped
Which of the following terraform subcommands could be used to remove

the lock on the state for the current configuration?

- unlock
- state-unlock
- Removing the lock on a state file is not possible
- force-unlock
  (Correct)

Explanation

terraform force-unlock removes the lock on the state for the current configuration.
https://www.terraform.io/docs/commands/force-unlock.html

Question 44: Skipped
What is the result of the following terraform function call?

> zipmap(["a", "b"], [1, 2])

- {
  "a",
  "b",
  "1",
  "2",
  }
- {
  "a" = 1
  "b" = 2
  }
  (Correct)
- [
  "a" = 1
  "b" = 2
  ]
- [
  "a",
  "b",
  "1",
  "2",
  ]

Explanation

zipmap constructs a map from a list of keys and a corresponding list of values. A map is denoted by { } whereas a list is donated by [ ].

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

Question 45: Skipped
Select all Operating Systems that Terraform is available for. (select five)

- Solaris
  (Correct)
- macOS
  (Correct)
- Windows
  (Correct)
- Unix
- FreeBSD
  (Correct)
- Linux
  (Correct)

Explanation

Terraform is available for macOS, FreeBSD, OpenBSD, Linux, Solaris, Windows

https://www.terraform.io/downloads.html

Question 46: Skipped
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 terraform plan and state files.

- True
- False
  (Correct)

Explanation

Currently, Terraform has no mechanism to redact or protect secrets that are returned via data sources, so secrets read via this provider will be persisted into the Terraform state, into any plan files, and in some cases in the console output produced while planning and applying. These artifacts must, therefore, all be protected accordingly.

Question 47: Skipped
Which of the following commands will launch the Interactive console for Terraform interpolations?

- terraform console
  (Correct)

- terraform
- terraform cli
- terraform cmdline

Explanation

The terraform console command provides an interactive console for evaluating expressions.

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

Question 48: Skipped
Which of the following is not a valid Terraform string function?

- format
- replace
- join
- tostring
(Correct)

Explanation

tostring is not a string function, it is a type conversion function. tostring converts its argument to a string value.
https://www.terraform.io/docs/configuration/functions/tostring.html

Question 49: Skipped
When multiple arguments with single-line values appear on consecutive lines at the same nesting level, HashiCorp recommends that you:

- align their equals signs

```
ami          = "abc123"
instance_type = "t2.micro"
```

(Correct)

- place a space in between each line

```
type = "A"

ttl = "300"

zone_id = aws_route53_zone.primary.zone_id
```

- put arguments in alphabetical order

```
name = "www.example.com"
records = [aws_eip.lb.public_ip]
type = "A"
ttl = "300"
zone_id = aws_route53_zone.primary.zone_id
```

- place all arguments using a variable at the top

```
ami = var.aws_ami
instance_type = var.instance_size
subnet_id = "subnet-0bb1c79de3EXAMPLE"
tags = {
Name = "HelloWorld"
}
```

Explanation

HashiCorp style conventions suggest you that align the equals sign for consecutive arguments for easing readability for configurations

```
ami          = "abc123"
instance_type = "t2.micro"
```

Question 50: Skipped
Which Terraform command will check and report errors within modules, attribute names, and value types to make sure they are syntactically valid and internally consistent?

- terraform fmt
- terraform format
- terraform validate
(Correct)
- terraform show

Explanation

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 the correctness of attribute names and value types.

Question 1: Skipped

You want to provision a resource using a community provider with Terraform version 0.13.3. You have been told that you need to manually install the provider plugin manually as it is a community provider.

True or False?

- True
- False
  (Correct)

Explanation

with 0.13 both Terraform certified and community provider can be downloaded automatically

Question 2: Skipped

You have started to use Terraform conditional dependencies as expressions in your Terraform configuration to add a condition to chose a compute instance type based on provider cost tier in order to manage your CAPEX. You are trying to find the correct syntax for your conditional expression.

Please select the correct answer:

-

```
machine_type = var.gcp-tier != "free-tier" ? var.gcp-machinetype[0] : var.gcp-machinetype[1]
```

(Correct)

-

```
machine_type = var.gcp-tier != "free-tier" | var.gcp-machinetype[0] , var.gcp-machinetype[1]
```

-

```
machine_type = var.gcp-tier != "free-tier" // var.gcp-machinetype[0] : var.gcp-machinetype[1]
```

-

```
machine_type == var.gcp-tier != "free-tier" ? var.gcp-machinetype[0] : var.gcp-machinetype[1]
```

Explanation

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

The syntax of a conditional expression is as follows:

condition ? true_val : false_val

Question 3: Skipped

You Team leader has been reviewing your Terraform code and has required you to rewrite your code to a canonical format and style. What Terraform command would you use to do that?

Please select the correct answer

- terraform format canonical
- terraform fmt
  (Correct)
- terraform config format
- terraform output fmt

Question 4: Skipped

You have decided to no longer manage a resource in your infrastructure provider but do not want to delete that resource block as you may want to use at a later stage. What is the best way to accomplish that?

please chose the correct answer:

-

```
//
resource "aws_s3_bucket" "mys3prod" {
bucket = "tf-dean-bucket-prod"
acl    = "public-read-write"


tags = {
Name        = "bucket"
Environment = "Prod"
}
}
##
```

-

```
##
resource "aws_s3_bucket" "mys3prod" {
bucket = "tf-dean-bucket-prod"
acl    = "public-read-write"


tags = {
Name        = "bucket"
Environment = "Prod"
}
```

```
}
##

/*
resource "aws_s3_bucket" "mys3prod" {
bucket = "tf-dean-bucket-prod"
acl    = "public-read-write"


tags = {
Name        = "bucket"
Environment = "Prod"
}
}
/*

/*
resource "aws_s3_bucket" "mys3prod" {
bucket = "tf-dean-bucket-prod"
acl    = "public-read-write"


tags = {
Name        = "bucket"
Environment = "Prod"
}
}
*/
```

(Correct)

Explanation

use /* */

Question 5: Skipped

You have opted to use Terraform cloud and are in the process of migrating your local backend to Terraform cloud. You have created an organization in Terraform cloud but did not create any workspace. what would happen when you do a terraform init?

```
terraform {

backend "remote" {
organization = "tf-cloud-lab"

workspaces {
name = "dev"
}
}
}
```

please select the correct answer:

- Terraform init will succeed but no workspace will be automatically created in Terraform cloud
- Terraform init will fail with an invalid terraform backend configuration error.
- The Terraform init will succeed and the workspace dev will be automatically created in Terraform cloud
  (Correct)
- The terraform init would fail as no workspace dev was created in Terraform cloud

Explanation

organization has to be preconfigured but workspaces will be automatically created

Question 6: Skipped

You have been using the language json to write various type of automation templates and would like to use json as well to write your Terraform configuration files. Which file extension would you use for your Terraform configuration to accomplish that?

please select the correct answer:

- .tfjson
- .tf.json
  (Correct)
- .json
- .tf

Explanation

https://www.terraform.io/docs/configuration/index.html under code organization

json can be used instead of HCL but required the .tf.json extension. The HCL uses .tf

Question 7: Skipped

You are reviewing your state file and you have noticed the lineage id. What is the role of the lineage id specified in the state file?

please select the correct answer:

- a unique id number in the state to identify the infrastructure provider hosting the Terraform managed resources.
- State files are versioned as Terraform is immutable and the lineage indicates the latest version number of the state file.

- a unique identifying number associated with the state file and generated at the creation of the state file.
  (Correct)
- a random id number generated at the creating of the state file for audit logging purpose.

Explanation

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

unique id to identify the state file. Used to compare state files during a terraform state push

Question 8: Skipped
You need to replicate resources with identical configuration. You trying to understand the difference between count and for_each.

Please select the best correct answer explaining the difference:

- for_each is used only in dynamic blocks.
- count uses a numerical index to address resources whereas for_each uses a string index for each resources. for_each only supports values of type maps and lists.
- count is to be used when the resources configuration are identical whereas fo_each is be used when resources configuration are not identical and arguments require distinct values.
  (Correct)
- count is to be used when the resources configuration are not identical and some arguments require different values whereas fo_each is be used when resources are using identical configuration.
- count and for_each have the same role and are used to replicate resources of the same type. for_each was introduced in Terraform version 0.12

Explanation

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

Question 9: Skipped
You have initialized your root module and configured the provider as google. You have noticed that the name of the provider plugin in the .terraform directory is. Which Terraform version are you using?

please select the correct answer:

818. registry.terraform.io/hashicorp/google/2.0.0/darwin_amd64/terraform-provider-google_v2.0.0

- 0.12.0
- 0.12.29
- 0.11.1
- 0.13.2
  (Correct)

Explanation

https://www.terraform.io/upgrade-guides/0-13.html

The registry.terraform.io above is the hostname of the registry considered to be the origin for this provider. The provider source address hashicorp/google is a shorthand for registry.terraform.io/hashicorp/google, and the full, explicit form is required for a local directory.

Question 10: Skipped
You have published module in the Terraform cloud private registry and would like to allow your organization users to search for module using the provider name. which valid provider name would you use to accomplish?

please select the correct answer:

- ALIBABA
- GCP
- GOOGLE
  (Correct)
- AZURERM

Explanation

AWS, GOOGLE and AZURE are the only supported names

Question 11: Skipped
Your new job as the Terraform admin is to use Terraform to manage resources in a multi cloud environment. On your first week you have noticed that users are using various version of the Terraform google cloud plugin. You have decided to have all users to configure the google cloud plugin version to any version between 3.0 and 3.9 would be acceptable

please select the BEST correct answer:

-

```
provider "google" {
region = "us-east4"
zone = "us-east4-a"
version = "= 3.9"
credentials = file("~/mykeys.json")
pro
```

```
terraform {
required_providers {
google = {
version = ">= 3.0 <= 3.9"
}
}
}
```

```
provider "google" {
region = "us-east4"
zone = "us-east4-a"
version = "~> 3.0"
credentials = file("~/mykeys.json")
project = "terraform-123456"
}
```

```
terraform {
required_providers {
google = {
version = "~> 3.0"
}
}
}
```

(Correct)

Question 12: Skipped

The Ops team has declared the Terraform variable value as a list of ec2 instance types using capital letters. Knowing that aws requires instance types to be declared using lower case, how can you iterate over the list and change the capitalization of each element of that list using the for expression?

```
variable "type" {
default = ["T2.MICRO","T2.LARGE","T2.NANO"]
}
```

- {for e in var.type : lower(e)}
- [for e in var.type : lower(e)]
  (Correct)
- [{for e in var.type : lower(e)}]
- [for s in var.type ? lower(s)]

Explanation

the for expression for a list is enclosed in **[]** with syntax [for e in <list_of_strings> : some code]

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

Question 13: Skipped

Commenting your code is highly recommended by your team leader and is asking you to add comments to explain your code. What is the default comment style in Terraform?

please chose the correct answer:

- //
- #
  (Correct)
- //*
- /**/

Explanation

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

The # single-line comment style is the default comment style and should be used in most cases. Automatic configuration formatting tools may automatically transform // comments into # comments, since the double-slash style is not idiomatic.

Question 14: Skipped

Your are planning to install Terraform and would like to confirm which Operating Systems are supported by Terraform

please select 4 correct answers

- Unix
- Linux
  (Correct)
- freeBSD
  (Correct)
- MirBSD

Solaris
(Correct)

- Windows
(Correct)

Question 15: Skipped
Your company have asked you to backup your Terraform execution plan. What would the recommended way to accomplish that?

Please select the correct answer:

- terraform plan -out="awsplan"
(Correct)
- terraform plan -var-file="awsplan"
- terraform plan > awsplan
- terraform plan -backup="awsplan"

Explanation

terraform plan > awsplan would backup the output of the plan but thats not is a backup that you can use with the terraform plan command

The correct way is to use the -out flag

Question 16: Skipped
You have learned about the use meta-arguments in your Terraform configuration.

Which are valid Terraform meta-arguments in Terraform from the list below

Please select 3 correct answers:

- source
(Correct)
- count
(Correct)
- providers
- depend_on
- lifecycle
(Correct)
- version

Explanation

https://www.terraform.io/docs/configuration/resources.html. Please be careful about spelling.

its depends_on not depend_on

Question 17: Skipped
The benefit of the terraform taint command is that it will automatically destroy and recreate a Terraform managed resource in the infrastructure provider.

True or False?

- True
- False
(Correct)

Explanation

taint will only flag a resource for destruction and recreate at the next terraform apply

Question 18: Skipped
You have created a few resources in google cloud using Terraform. You have noticed that a state file was created automatically and you are concerned about doing a backup as it is a critical file. Will a backup state file be created automatically at the time of the creation of the state file?

True or False?

- True
(Correct)
- False

Explanation

Yes..terraform.tfstate.backup

Question 19: Skipped
You have heard that the recommended way by Terraform to execute a script on a Google compute instance is to always use a remote-exec provisioner.

True or False

- True
- False
(Correct)

Explanation

Terraform recommend to use provisioners only as a last resort.

https://www.terraform.io/docs/provisioners/index.html under the How to use provisioners

Question 20: Skipped
Your organization is requiring that you download the latest provider plugin every time you are creating an execution plan and apply it. Which command would you use to accomplish that?

please select the correct answer:

- terraform init -upgrade
  (Correct)
- terraform init -update
- terraform plugin -latest
- terraform plan -upgrade

Explanation

The latest plugin version is downloaded by default only during the first terraform init.

Question 21: Skipped
You are reviewing your state file and trying to understand the use of its serial number.

please select the correct answer:

```
849. {
850. "version": 4,
851. "terraform_version": "0.12.29",
852. "serial": 5,
853. "lineage": "7ccd8971-2e1a-72b5-13ee-c9aeb63a22b5",
854. ...
855. }
```

- The serial increments every time a terraform apply is successful.
  (Correct)
- The serial number is used to version the state file in Terraform cloud for state file versioning
- The serial number is a number used a unique identifier for the state file.
- The serial number will be used to determine whether 2 states files can be successfully merged with terraform state mv command with remote state.

Explanation

The serial number will be used to determine whether 2 states files can be successfully merged with the terraform state push command and not mv.

Question 22: Skipped
You have migrated to Terraform cloud but would like to use CLI driven run for your workspaces to perform Terraform operations. What is a requirement to do that?

please select the correct answer:

- enter terraform plan and apply and view output in Terraform Cloud.
- terraform login at the CLI and use API key generated in Terraform cloud. Enter Terraform plan and apply at the local CLI. The run output of will be streamed on your CLI from Terraform cloud
  (Correct)
- terraform login at the CLI and enter your username and password
- terraform login at the CLI and use API key generated in Terraform cloud. Perform terraform plan and apply locally at the CLI on your local machine

Question 23: Skipped
You wanted to test the Terraform refresh command. What will happen when running the Terraform refresh command?

Please select the correct answer

- It will update the state file with any configuration change done manually on the Terraform managed resource in your real world infrastructure provider and apply the changes by modifying the real world infrastructure configuration.
- It will only update the state file with any configuration change of the Terraform managed resource in your real world infrastructure provider
  (Correct)
- It will only update the state file with any change done manually in the real world infrastructure provider hosting your Terraform managed resources.

Question 24: Skipped
Terraform is an infrastructure provisioning and configuration management tool.

True or False?

- True
-

False
(Correct)

Explanation

terraform is only an infrastructure provisioning tool.

Ansible is an example of configuration management.

Question 25: Skipped
You have configured an EC2 instance resource block as follow. Will the terraform apply be successful?

True or False?

```
resource "aws_instance" "ec2-prod" {
ami = data.aws_ami.ami.id
instance_type = "t2.micro"
tags = {
Name = "ec2-vm"
}
}
}
```

- True
- False
(Correct)

Explanation

always count your curly braces at the end to constrain the code in the resource block. one too many at the end

```
resource "aws_instance" "ec2-prod" {
  ami = data.aws_ami.ami.id
  instance_type = "t2.micro"

  tags = {
    Name = "ec2-vm"
  }

}
```

Question 26: Skipped
You are reviewing the Terraform configuration to create a Google compute instance template. What is the local name of the instance? Please select the correct answer:

```
resource "google_compute_instance_template" "dev_template" {
name = "dev-instance-template"
machine_type = var.gcp-tier != "free-tier" ? var.gcp-machinetype[0] : var.gcp-machinetype[1]
tags = ["apache"]
metadata_startup_script = var.linuxtype == "ubuntu" ? var.debian-script : var.centos-script

disk {
source_image = var.linuxtype == "ubuntu" ? var.gcp-image[0] : var.gcp-image[1]
}

network_interface {
network = google_compute_network.webnetwork.name
subnetwork = google_compute_subnetwork.websubnet.name
access_config {
}
}
```

- apache
- dev-instance-template
- dev_template
(Correct)
- dev_instance_template

Explanation

resource "resource_type" "local_name" {}

Question 27: Skipped
You have decided to rename your state file to acme_state.tfstate in your root module. The terraform apply is not able to read the state file. How would resolve the problem?

Please select the correct answer:

- Renaming a state file is not supported by Terraform
- terraform apply -state="./acme_state.tfstate"
(Correct)
- terraform apply -state-output="./acme_state.tfstate"
- terraform apply -var-file="./acme_state.tfstate"

Question 28: Skipped
You have to been requested by your company to use the pessimist constraint operator to set the allowed Terraform version to enforce terraform version consistency. Which operator would you use to accomplish that?

Please select the correct operator in the answers below:

-

- !=
- ?
- >=
- ~>
(Correct)

Explanation

~>: Allows the specified version, plus newer versions that only increase the *most specific* segment of the specified version number. For example, ~> 0.9 is equivalent to >= 0.9, < 1.0, and ~> 0.8.4, is equivalent to >= 0.8.4, < 0.9. This is usually called the pessimistic constraint operator.

Question 29: Skipped
You have decided to use local workspaces in your root module. You have created 2 workspaces dev and prod. What directory is automatically created at the creation of a workspace

Please select the correct answer:

- .terraform.tfstate/
- .terraform.tfstate.d/
(Correct)
- .terraform.d/states/
- .terraform/workspaces/

Question 30: Skipped
You have decided to enforce best practices in your Terraform configuration and you have constrained the variable value type as follow:

please select the correct value for this variable:

```
884. variable "ingress" {
885. description = "ingress ports"
886. type = list(map(any))
887. default = "value"
888. }
```

- [port1=80,port2=22,port3=443,port4=53]
- {[port1=80,port2=22,port3=443,port4=53]}
- [{port1=80,port2=22,port3=443,port4=53}]
(Correct)
- [80,22,443,53]

Explanation

map is constrained with {} and a list is constrained with [] so a map within a list is [{}]

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

Question 31: Skipped
Users a large bank have been giving IAM accounts and are deploying S3 buckets and EC2 instances in AWS. You are concerned about the security of your cloud posture and defined several sentinel policies to enforce compliance to secure configurations. You have been tasked to create enforcement levels for each policy. How would you accomplish that.

please select the Terraform recommended correct answer:

- using the enforcement block in the policy.sentinel file in VCS connected to your Terraform cloud workspace
- define enforcement level for each policy in a sentinel.hcl file in the VCS connected to your Sentinel policyset.
- Individual Sentinel individual policy settings using the enforcement level drop down menu
- using the enforcement block in the policy.hcl file in the VCS connected to your Terraform cloud workspace
- define enforcement level in each policy block in a sentinel.hcl file in the VCS connected to your Sentinel individual policy.
(Correct)

Explanation

Sentinel individual policies are deprecated. using policysets and defining enforcement level in the policy block is the recommended approach by Terraform.

Question 32: Skipped
You are a managing the infra team in your company and have required that all users in your team have the Terraform settings block with the required_version argument value as ~>0.12.1

Which Terraform version would you all the users be allowed to use for Terraform operations on your local machine?

please select correct answers:

-

any version up to 0.12.1

- any version between 0.12.1 and 0.12.29 inclusive
(Correct)
- any version from 0.12.1 and above
- only version 0.12.1
- any version between 0.12.1 and 0.13.0 inclusive.

Explanation

~> 0.12.1 is equivalent to ">= 0.12.1, < 0.13.0"

Question 33: Skipped
You have configured your terraform provider with multiple regions of aws. You need to deploy the EC2 instance VM-dev in region us-east-2. You are looking at the documentation to find be the configuration to accomplish that. Your provider configuration is as follow:

```
provider "aws" {
region = "us-east-2"
alias = "ohio"
}

provider "aws2" {
region = "us-east-1"
}

provider "aws3" {
region = "us-west-1"
alias = "california"
}
```

please select the correct answer:

-
```
resource "aws_instance" "ec2-prod" {
ami = data.aws_ami.ami.id
instance_type = "t2.micro"
availability_zone = "us-east-2b"
provider = aws.ohio

tags = {
Name = "VM-dev"
}
```
-
```
resource "aws_instance" "VM_dev" {
name = "VM-dev"
ami = data.aws_ami.ami.id
instance_type = "t2.micro"
availability_zone = "us-east-2b"
provider = aws.ohio
}
```
-
```
resource "aws_instance" "VM-dev" {
ami = data.aws_ami.ami.id
instance_type = "t2.micro"
availability_zone = "us-east-2b"
provider = aws.ohio
}
```
(Correct)
-
```
resource "aws_instance" "ec2-prod" {
name = "VM-dev"
ami = data.aws_ami.ami.id
instance_type = "t2.micro"
availability_zone = "us-east-2b"
provider = aws.ohio
}
```

Explanation

resource block argument provider = <provider>.<alias_name>

Question 34: Skipped
You have configured the following Terraform variable named ami_id and would like to refer to that variable as the expression for the ami argument in your aws instance resource block.

```
variable "ami_id"{
default = "ami-123456"
}

resource aws_instance "ec2-prod" {
ami = ??
...
}
```

Please select the correct answer:

- ${ami-123456}
- ${var.ami}
- "ami-123456"
- var.ami
(Correct)

Explanation

correct syntax is var.<variable_name>

Question 35: Skipped

Your team has been using Terraform to provision and manage resources in AWS. various team members have been installing different versions of Terraform which is starting to cause problems with configuration consistency. You have decided to require that the same Terraform version be configured as 12.1.

Please select the correct answer:

-

```
terraform {
required_version = {
version = "12.1"
}
```

-

```
terraform {
required_version {
version = "12.1"
}
```

-

```
provider "aws" {
region = "us-east-2"
version = "12.1"
}
```

-

```
terraform {
required_version = "12.1"
}
```

(Correct)

Question 36: Skipped

You have tainted a Terraform managed resource but decided later than you do not want to taint it anymore.

How would you remove the taint from the resource?

Please select the correct answer:

- It is not possible to remove the taint from a resource
- terraform taint rm
- terraform untaint
  (Correct)
- terraform state untaint

Question 37: Skipped

You have realized that you would need to do some configuration management on some of your Terraform managed resources and decided to use the remote-exec provisioner in order to install an apache web server on you compute instance. What type of connections to the remote host are supported by this Terraform provisioner in order to accomplish that?

Please select 2 correct answers:

- rcp
- scp
- win
- ssh
  (Correct)
- winrm
  (Correct)

Question 38: Skipped

Your are in charge of managing the security configuration for your company Terraform managed resources provisioned in AWS. The security requirements for your security groups requires that all whitelisted ingress ports need to specified rather allowing all. You have discovered the benefit of using dynamic blocks for your security groups configuration but you are not sure which expression to use for the to_port and from_port arguments in content{}. Which expression would accomplish that?

```
resource "aws_security_group" "awsfw" {
name = "aws-fw"
vpc_id = aws_vpc.tfvpc.id

dynamic "ingress" {
for_each = toset([80,53,443,22,21])

content {
description = ingress.value.description
protocol = "tcp"
to_port = ??
from_port = ??
cidr_blocks = ["0.0.0.0/0"]
}
}
```

Please select the correct answers:

- ingress.value.key

(Correct)

- awsfw.value.key
- each.key
- awsfw.value.key
- ingress.value.value
(Correct)

Explanation

dynamic blocks section under

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

Question 39: Skipped
You are configuring a remote-exec provisioner for your aws instance. You want to reference the public IP address of your aws instance in the provisioner configuration. Which expression for the host argument would you use to accomplish in this configuration?

```
resource "aws_instance" "ec2-prod" {
ami = data.aws_ami.ami.id
instance_type = "t2.micro"
availability_zone = "us-east-2b"
provider = aws.ohio

provisioner "remote-exec" {
connection {
host = ??

}
}
}
```

please select the correct answer

- self.public_ip
(Correct)
- aws_instance.ec2-prod.ip
- aws_instance.self.public_ip
- aws_instance.ec2-prod.public_ip

Question 40: Skipped
You are installing Terraform on your local machine. As Terraform is GO binary Terraform requires GO to be installed as a prerequisite for Terraform to be successfully installed in a local machine.

True or False?

- False
(Correct)
- True

Explanation

Go is not a prerequisite

Question 41: Skipped
You are new to Terraform but you have asked to deploy an application load balancer or ALB in AWS today using Terraform. You have downloaded the ALB module from the Terraform public registry. Which configuration block type would you use to use this ALB module as a child module?

please select the correct answer:

-
```
module {
  import = "path to ALB module"
}
```
-
```
module "alb" {
  source = "path to ALB module"
}
```
(Correct)
-
```
module {
  source = "path to ALB module"
}
```
-
```
module "alb" {
  source = "child module name"
}
```

Question 42: Skipped
Your users have been creating resources with Terraform cloud with insecure configuration. What tool would you use to enforce your configuration policies during Terraform runs?

please select the correct answer:

- Need to integrate an third party tool to Terraform cloud

- Sentinel
  (Correct)
- Vault
- Sensitive policysets

Question 43: Skipped

You started using Terraform data sources. How can you reference the data source to reference the ami id in the resource block below:

```
data "aws_ami" "ami_id" {
owners = ["amazon"]
most_recent = true

filter {
name = "name"
values = ["amzn2-ami-hvm-2.0.20200722.0-x86*"]
}
}

resource "aws_instance" "ec2-prod" {
ami =
instance_type = "t2.micro"
availability_zone = "us-east-2b"
}
```

Please select the correct answer:

- ami = ${aws_ami.ami_id.id}
- ami = "data.aws_ami.ami_id.id"
- ami = data.aws_ami.ami_id.id
  (Correct)
- ami = data_source.aws_ami.ami_id.id

Question 44: Skipped

Your organization has set new requirements with the use of modules as blueprints. You have added a new module in your root module. Which configuration block you will have to use in order to refactor your resource configuration file to use these modules as child modules.

Please select the correct block type:

- source {}
- resource {}
- source_module {}
- module {}
  (Correct)

Question 45: Skipped

You have a total of 5 Terraform managed resource in an infrastructure provider. You have to destroy one Terraform managed resource from the infrastructure. What are the correct commands to accomplish that?

Please select the correct answers:

- terraform destroy -target=<resource_address> then terraform apply
  (Correct)
- terraform destroy <resource_address> then terraform apply
  (Correct)
- terraform delete
- terraform plan -destroy <resource_address> then terraform apply
- terraform destroy

Explanation

You are destroying only one resource and thus the -target flag with the resource address is required to target that particular resource

Question 46: Skipped

Your company has decided to use Terraform as the infrastructure as code tool. They have to use Terraform to manage an ec2 instance in aws which was manually configured. The aws id of this EC2 is id-123456 How can they use Terraform to manage this ec2 instance?

Please select the correct answers:

- Use the terraform import id-123456 command to import the ec2 instance to Terraform.
- First create the resource configuration in Terraform then use the terraform import command to import the ec2 instance to Terraform.
- It is not possible to import manually created resources to Terraform
- use the terraform import command to automatically import the ec2 instance in Terraform.
- First create the resource configuration in Terraform then use the terraform import

aws_instance.ec2_dev.id command to import the ec2 instance to Terraform.
(Correct)

Question 47: Skipped

You are leading a Terraform project to provision a complex infrastructure in Azure. You have created a directory for the project and have added all the Terraform configuration files in the directory. You are ready to provision all the resources declared in the configuration. What is the first Terraform command to accomplish that?

Please select the correct answer:

- Terraform initialize
- Terraform plan
- Terraform init
(Correct)
- Terraform validate

Explanation

you need to first initialize your root module.

Question 48: Skipped

Terraform ensures that the state file is always encrypted at rest in a local backend by default

True or False?

- False
(Correct)
- True

Explanation

state file not encrypted in local backend by defaut

Question 49: Skipped

You have migrated your local backend to S3 as your standard backend to store the state remotely. You have then provisioned resources using the S3 backend. You forgot that you had a local backup state file and now need to merge that state file to the state file stored remotely in S3. What is one of Terraform requirement for the merge to be successful?

Please select the correct answer:

- the merge will always be successful since this was a successful migration to S3
- lower state file lineage id number on the remote state file.
- lower state file version number on the local state file
- the local state file and the remote state file need to have the same lineage id
(Correct)

Explanation

https://www.terraform.io/docs/backends/state.html under **Manual State Pull/Push**

lineage id of both state files have to be same

Question 50: Skipped

Your company has decided to move to Terraform cloud. You have been tasked to migrate your local backend to Terraform cloud. You had installed Terraform version 0.12.29 on your local machine and created all your configuration with that version. Which Terraform version will Terraform cloud use in Terraform runs by default once you have completed the migration?

Please select the correct answer:

- version 0.12.29
- The latest Terraform version
(Correct)
- You will have to set the version in Terraform cloud prior to queue a run in the workspace configuration to queue a Terraform run.
- The latest 0.12.x version.

Explanation

The default is latest and you can change to your preferred that in the workspace settings

Question 1: Skipped

You have decided to contribute to the Terraform community and would like to publish your module for the AWS EC2 compute instance to the Terraform public registry. What are the requirements defined by Terraform to publish a module?

Please select 3 correct answers:

- add Git tags to version your modules using a semantic version
(Correct)
-

use GitHub
(Correct)

- Use any supported VCS like GitHub, bitbucket or Gitlab
- name your repository module-aws-ec2
- name your repository terraform-aws-ec2
(Correct)
- add tags to version your modules using the name of your module

Explanation

https://www.terraform.io/docs/registry/modules/publish.html **requirements**

its only github, git tags using semantic versioning like v1.0.2 and terraform-<provider>-name

Question 2: Skipped
You have migrated your local backend to Terraform cloud. You are creating a workspace and chose to use a VCS workflow. Which VCS are supported by Terraform cloud.

Please chose 3 correct answers:

- Google Artifact Registry
- Bitbucket
(Correct)
- GitHub
(Correct)
- Azure Devops
(Correct)
- S3
- AWS CodeCommit

Question 3: Skipped
Terraform crashed after running the terraform apply command. Where will the Terraform crash file be stored by default?

please select the correct answer:

- your disk home directory
- in the .terraform directory in your root module
- no crash file is generated by default and only outputted to CLI at the time of the crash
- In the directory which you configured in the terraform setting block.
- the root module directory
(Correct)

Question 4: Skipped
Your CIO is asking you to outline the benefits of using Infrastructure as Code in the cloud. Please select the correct reason to start using Infrastructure as Code

- Infrastructure as code is a great tool to create and manage resource in an infrastructure provider as it uses a high level coding language which can be shared and reused. However it cannot be versioned
- Infrastructure as code is a great tool to create and manage resource in an infrastructure provider as it uses a high level coding language which can be shared and reused. The code can be used as templates that can be versioned in VCS.
(Correct)
- Infrastructure as code is great tool to deploy applications as it uses a high level coding language which can be versioned and can be reused.
- Infrastructure as code is a great tool to create infrastructure as it uses a low level coding language which can be versioned and shared across teams within the company

Explanation

https://www.terraform.io/intro/index.html

Terraform does not deploy apps but provision infrastructure.

it can be versioned with VCS

it is not a low level language but high level language

Question 5: Skipped
You have decided to use terraform to manage resources in Google cloud. What would be the required step to do that?

- Create a plugins subdirectory in the .terraform directory and configure the google provider. Terraform will download automatically the google plugin binary from the Terraform registry after initializing your Terraform root module
- Configure google as a provider and Terraform will automatically create a .terraform directory and download the google binary when you initialize your Terraform root module

(Correct)

- Configure google as a provider and Terraform will automatically create a .terraform directory and download the google binary.
- Create a plugins subdirectory in the .terraform directory and download the google plugin binary from the Terraform registry. Configuring google as a provider will allow you to start deploying resources in Google cloud.

Question 6: Skipped

Sentinel policysets apply for all the organizations created in Terraform cloud.

True or False?

- False
(Correct)
- True

Explanation

Sentinel Policysets are configured within one organization but can apply to all or selected workspaces

Question 7: Skipped

The order of action of the terraform taint command in the execution plan cannot be changed.

True or false?

- true
- false
(Correct)

Explanation

Yes it can be changed with the lifecycle meta argument using create_before_destroy = true

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

Question 8: Skipped

Using the Terraform remote provisioners to install an apache web server on your GCP compute instance is not recommended as best practice by Terraform.

True or false?

- true
(Correct)
- false

Explanation

provisioners should be used only as last resort

https://www.terraform.io/docs/provisioners/index.html

Question 9: Skipped

You Terraform code has been reviewed by the team. You have requested by your supervisor to use Terraform variables as expressions instead of literal values for the arguments in the resource blocks. What would be correct expression?

Please select the correct answer:

- variable<variable_name>
- var<variable_name>
(Correct)
- vars<variable_name>
- tfvar<variable_name>

Explanation

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

Question 10: Skipped

You are configuring an aws autoscaling group resource block which has an implicit dependency with the aws launch configuration. The id of the launch configuration need to be declared with the launch_configuration argument in the autoscaling group resource block. The local name of the launch configuration resource block is aws_launch

What is the correct expression for the launch_configuration argument?

please select the correct answer:

- resource.aws_launch_configuration.aws_launch.id
- resource.aws_launch_configuration.id
- resource.aws_launch.id
- aws_launch_configuration.aws_launch.id
(Correct)

Explanation

reference to another resource attribute uses the following syntax

resource_type.name.attribute

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

Question 11: Skipped
You have created using Terraform an EC2 instance with a user data script to install your apache web server. The customers have been accessing the web application. You have been asked to change the web server from apache to nginx. What would be the best way to change to do that without impacting the user access to the web server?

Please select the best answer:

- update the user data script to install nginx and taint the EC2 resource.
- update the user data script to install nginx and perform a terraform apply
- create a new EC2 instance with a new resource block and a user data script using nginx then destroy the EC2 instance with the apache server. forward the new IP address to your customers to access the nginx web server.
- update the user data script and add a lifecycle argument to prevent disruption then do a terraform apply
  (Correct)

Explanation

Updating a user data script cause a resource to be destroyed and created again causing disruption to the users.

best approach is using the lifecycle argument with create_before_destroy

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

Question 12: Skipped
You want to install an nginx server on your Google compute instance that you are going to create using a resource block with Terraform with all required arguments. Which would be the correct argument to add the script to the resource block?

please select the correct answer:

-
```
resource "google_compute_instance" "VM_test" {
name = "VM-test"
...
metadata_startup_script = <<-EOF
sudo apt-get install nginx
EOF
...
}
```
  (Correct)
-
```
resource "google_compute_instance" "VM_test" {
name = "VM-test"
...
user_data = <-EOF
sudo apt-get install nginx
EOF

}
```
-
```
resource "google_compute_instance" "VM_test" {
name = "VM-test"
...
script_data = <<-EOF
sudo apt-get install nginx
EOF
...
}
```
-
```
resource "google_compute_instance" "VM_test" {
name = "VM-test"
...
user-data = <<-EOF
sudo apt-get install nginx
EOF
...
}
```

Explanation

https://www.terraform.io/docs/providers/google/r/compute_instance.html

Question 13: Skipped
You trying to explain to your team why Terraform is immutable. Please chose the best reason why?

Please choose the correct answer below:

- Immutability allow a user to track all changes made to a resource configuration in order to manage complex infrastructure deployments.
- Immutability reduces configuration drift and makes it easier to roll back to a previous version if an upgrade fails.
  (Correct)
- Immutability is optional as it can cause accidental deletion of resources in an infrastructure. It can be disabled in the Terraform settings

- Immutability requires the user to destroy all resources created in a provider if a change is needed in one of those resource in order to meet the configuration desired end state.

Explanation

https://www.hashicorp.com/resources/what-is-mutable-vs-immutable-infrastructure

immutability doesn't require you to track configuration changes as the configured cannot be mutated or changed and thus immutable.

immutability require the creation a new resource with a desired end state. It does not require to destroy any resources for it to be immutable.

no config drift as there is no change in config but rather the creation of a new instance with the new desired end state

Question 14: Skipped
You are working to create an EC2 instance resource block and need to find the latest ami id for an ubuntu image for the ami argument. What would be the most efficient way to get the needed ami id?

note that the ami is the amazon machine image .

Please select the correct answer:

- use the <aws ec2 ami -latest >command
- use Terraform Data sources
(Correct)
- go to the aws management console under the EC2 service menu and search for the ami id under the ubuntu operating system
- use Terraform Data resources

Explanation

Using Data sources in terraform is the most efficient way to fetch any data from a provider.

Question 15: Skipped
You have decided to test Terraform as infrastructure as code tool because of its plugin based architecture. Please chose the accurate way to describe the plugin based architecture

Please select the correct answer:

- Terraform use one plugin for all the providers in order to be platform agnostic and support as many clouds as possible.
- The Terraform core binary includes the plugin for a provider which makes easy to provision and manage resources in that infrastructure provider.
- the Terraform provider plugin is a GO binary that allows Terraform to provision resources in many infrastructure provider
(Correct)

Question 16: Skipped
Your company has requested that you rename the state file to myacme.tfstate and back it up. Is it possible to rename the state file?

True or false?

- False
- True
(Correct)

Question 17: Skipped
Your team has been using Terraform and installing different versions of Terraform. What is the Terraform block argument to ensure that all your team members are using the same version?

please select correct answer:

-
```
required_version = {
"value"
}
```
-
```
required_version {
version = "value"
}
```
- required_version = "value"
(Correct)
- version = "<value>"

Explanation

```
terraform {
  required_version = ""
}
```

Question 18: Skipped
You have tasked to chose an Infrastructure as code tool for your organization and you have decided on Terraform. What can the best reason for using Terraform. Chose 2 correct answers below:

- Terraform uses a human readable high level coding language in any provider. It is also immutable and idempotent
(Correct)
- Terraform uses a HCL or high coding level to provision and manages resources in any provider. It is also immutable, idempotent and declarative.
- Terraform uses an easy to use high level coding language to provision and manage resources in only major cloud providers. It is also immutable, idempotent and declarative
- Terraform uses machine editable language that also is immutable and idempotent and declarative.
(Correct)

Explanation

Terraform is a human readable and machine editable language

it is idempotent, imperative and declarative

Question 19: Skipped
You have been reviewing the execution plan prior to applying it. Some resource will have to destroyed, updated and created. What are valid symbols indicating the actions to be taken on resources in an execution plan

Please select 3 correct answers:

- <=
(Correct)
- >=
- ~
(Correct)
- =!
- -/+
(Correct)
- ++

Question 20: Skipped
data security has become a critical requirement for your company and you have decided to use Terraform Vault.

Terraform will encrypt credentials and sensitive value in the state file.

True or false?

- False
(Correct)
- True

Question 21: Skipped
You have discovered the benefit of using Terraform variables and you are working to create a variable block named aws_region in order to set the region as us-east-2 in the aws provider block.

What is the correct argument to use in the aws_region variable block to declare the variable value as us-east-2?

Please select the correct answer:

- value = "us-east-2"
- default = "us-east-2"
(Correct)
- region = "us-east-2"
- default_value = "us-east-2"

Question 22: Skipped
You have been tasked to write the Terraform code to deploy four EC2 instances using the same configuration and arguments with identical values. What would be most efficient way to do without repeating your code?

please select the correct answer:

- use the count.index = 4 in one EC2 resource block
- use the for_each argument using for_each=tset([0,1,2,3]) in one EC2 resource block
- write a EC2 resource block and copy paste four times in your configuration
- use count = 4 in the one EC2 resource block
(Correct)

Explanation

for_each is best used when EC2 configs are not identical and require distinct values for their arguments.

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

Question 23: Skipped
What is the type of code language used in the state file?

please select the correct answer:

- HCL
- YAML
- JSON
  (Correct)
- GO

Question 24: Skipped
Terraform variables values declared in the terraform.tfvars file will override the values declared in the variable blocks.

True or False?

- False
- True
  (Correct)

Explanation

variable block declare default value whereas .tfvars declare actual value and will always take precedence.

Question 25: Skipped
You have been tasked to allow the team to install any of Terraform version 0.12 minor releases. What would be best configuration to accomplish that?

please select the best correct answer:

- *version = "-> 0.12.0"*
- version = ">= 0.12.0 < 0.13.0"
- version = "~> 0.12.0"
  (Correct)
- version = "~ 0.12.0"

Explanation

version = ">= 0.12.0 < 0.13.0" is correct but not the best answer. Using the pessimist constraint operator ~> is the best approach.

Question 26: Skipped
You are struggling to find the right Terraform function to convert your list of ingress ports into a set and use it as the expression for the for_each argument in your security group dynamic block. What Terraform function would you use in order to convert the list of port numbers into a set?

please select correct answer:

- set()
- tolist()
- toset()
  (Correct)
- listtoset()

Explanation

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

Question 27: Skipped
You have been tasked to compare Terraform to other infrastructure as code tools in the market. What are valid examples of other infrastructure as code solution. Please chose 4 valid answers

- Heat from Openstack
- Chef
  (Correct)
- Azure ressource management templates
  (Correct)
- Ansible
  (Correct)
- Cloud formation templates from Google
- Alicode from Alibaba

Explanation

Cloud formation template is from AWS and not Google

Alicode doesn't exist

Question 28: Skipped
You have created using Terraform an EC2 instance with a user data script to install your apache web server. However You are unable to access your web server and after much troubleshooting, you have identified the issue as a missing operator in your user data script. How can you trigger the script to run again once you have added the missing operator to your user data script?

Please select the best answer:

- add the missing operator to user data argument in the EC2 resource block and do a terraform apply
(Correct)
- taint the EC2 resource
- add the missing operator to the user data in the EC2 resource block and reboot your EC2 with the AWS cli or AWS management console.
- ssh to EC2 instance and run the script at the EC2 cli prompt.

Explanation

Terraform managed resources should be managed by Terraform.

Question 29: Skipped
You trying to figure the role of the terraform provider plugins in Terraform.

Please chose 2 correct answers below

- create, read, update and delete terraform resources in an infrastructure provider.
(Correct)
- Use RPC calls to an infrastructure provider to create and manage resources via Terraform operations.
- Use API calls to a provider to perform CRUD operations for Terraform managed resources in an infrastructure cloud provider
(Correct)
- Manage all the Terraform configuration files and ensure that the desired end state is always maintained at the infrastructure provider.

Question 30: Skipped
After reviewing your code, your team has requested that you configure an explicit dependency between a google compute instance named VM-test and a google target pool named lb. What would be the correct configuration in the google compute instance resource block?

please select correct answer:

-
```
resource "google_compute_instance" "VM-test" {
.....
depend_on = {
google_compute_target_pool.lb.id
}
}
```
-
```
resource "google_compute_instance" "VM-test" {
.....
depends_on {
google_compute_target_pool.lb
}
```
-
```
resource "google_compute_instance" "VM-test" {
.....
depends_on = {
google_compute_target_pool.lb
}
}
```
(Correct)
-
```
resource "google_compute_instance" "VM-test" {
.....
depends_on = google_compute_target_pool.lb
}
```
-
```
resource "google_compute_instance" "VM-test" {
.....
depends_on {
google_compute_target_pool.lb.id
}
}
```

Explanation

https://www.terraform.io/docs/configuration/expressions.html under explicit dependency

Question 31: Skipped
You read that Terraform configuration is declarative. How would explain why Terraform is declarative.

please select 2 correct answers:

- Terraform is declarative because it uses HCL code that is human readable and machine

editable

- terraform is declarative as it requires the user to understand the procedure to configure a resource in a provider in order to declare the desired end state in a way that is easy to understand.
- Terraform is declarative as it only declares the desired end state in a configuration for a resource.
  (Correct)
- Terraform is declarative as it does not require to declare all the steps needed to achieve desired end state.
  (Correct)
- Terraform is declarative as it use its human readable language to simply declare all the steps needed to achieve the desired end state

Explanation

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

Question 32: Skipped
You are trying to find the resource address for the aws-fw-dev security group in aws in order to fetch its id. What is the correct Terraform resource address for the security group configured as follow :

Please select the correct answer:

```
resource "aws_security_group" "awsfw" {
name = "aws-fw-dev"
vpc_id = aws_vpc.tfvpc.id

dynamic "ingress" {
for_each = local.ingress_config2

content {
description = ingress.value.description
protocol = "tcp"
to_port = ingress.value.port
from_port = ingress.value.port
cidr_blocks = ["0.0.0.0/0"]
}
}
```

- aws_security_group.awsfw
  (Correct)
- aws_security_group.aws-fw-dev
- resource.aws_security_group.aws-fw-dev
- resource.aws_security_group.awsfw

Explanation

the terraform resource address is:

resource_type.name

the name is the local name and not the name of the resource in the infra provider. Please be careful

Question 33: Skipped
You have realized that the benefits of using Terraform data sources to fetch the latest ami for amazon linux to use for the EC22 instance that you are planning to create using Terraform. ami being the amazon machine id.

Which terraform data source configuration would you use to accomplish that?

please select the correct answer:

-
```
data "aws_ami" "ami" {
owners = ["amazon"]
most_recent = true

filter {
name = "name"
values = ["amzn2-ami-hvm-2.0.20200722.0-x86*"]
}
}
```
(Correct)

-
```
data "aws_ami" "ami" {
owners = ["amazon"]
latest = true
filter {
name = "type"
values = ["amzn2-ami-hvm-2.0.20200722.0-x86*"]
}
}
```

-
```
data "aws_ami" "ami" {
owners = ["amazon"]

filter {
name = "name"
values = ["amzn2-ami-hvm-2.0.20200722.0-x86*"]
}
}
```

```
data "aws_ami" "ami" {
owners = ["amazon"]
most_recent = true

filter {
name = "architecture"
values = ["amzn2-ami-hvm-2.0.20200722.0-x86*"]
}
}
```

Explanation

you need to use the most_recent argument and be true

Question 34: Skipped
You have stored your state remotely on a standard backend to allow your team to collaborate. Two developers in your team are trying to perform Terraform apply at the same time. One of the developer is calling you that he is unable to apply as his Terraform system is locked. What would to suggest him to do?

Please select the best correct answer?

- terraform force_unlock Lock_ID
- terraform force-unlock Lock_ID
- Please wait till the lock is lifted and then try again to apply in order to prevent corruption of the state file.
  (Correct)
- terraform force_unlock Lock_ID
- terraform force_unlock LockID

Explanation

terraform force_unlock LockID is correct however...

The best answer is to wait as simultaneous applies can corrupt the state file and damage your ability to manage your terraform resources.

Question 35: Skipped
You have hired a new team member to learn terraform in order to write configuration for new resources that will be managed by Terraform. He forgot which file extension to use for terraform configuration files when using HCL code. What would be the correct file extension?

please select correct answer:

- .tconf
- .hcl
- .tf
  (Correct)
- .json

Question 36: Skipped
You have been using a terraform.tfvars file to assign values for your Terraform variables instead of declaring the values in all the variable blocks. You have decided to rename it to acme.tfvars based on corporate requirements. When performing a terraform apply you are asked the enter each value interactively at the CLI which is not convenient. The acme.tfvars file is located in your root module

What can you do to prevent terraform to ask for each variables value interactively at the CLI ?

Please select the correct answer:

- There is no other way. Inputting the variables value interactively at the CLI is the only way
- use the -var-file="./acme.tfvars" flag with the terraform apply command
  (Correct)
- use the --var-file="./acme.tfvars" flag with the terraform apply command
- Use the -var="acme.tfvars" with the terraform apply command

Explanation

use the -var-file="path" with terraform apply when renaming tfvars file

Question 37: Skipped
You are trying to configure a variable block and you trying to chose a valid local name for the variable. Which below are valid names:

please select the 3 correct answers:

- compute
  (Correct)
- locals
- filter
  (Correct)

- providers
- source
- lifecycles
(Correct)

Explanation

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

its lifecycle not lifecycles

Question 38: Skipped
You really liked the benefit of using dynamic blocks for your security group configuration. Is the use of dynamic blocks recommended by Terraform for all your security groups regardless on the number of whitelisted ports?

True or False?

- False
(Correct)
- True

Explanation

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.

Question 39: Skipped
You have used Terraform to provision a critical database in Azure using a local backend and you would like to ensure that no other team member can destroy your database accidentally. What would be the best way to accomplish that?

please select the correct answer

- use prevent_destroy argument in the database resource block to true
- Add a comment to the resource block for this database stating that this database cannot be deleted no matter what
- use the lifecycle meta argument in the database resource block and configure the prevent_destroy argument to true.
(Correct)
- Use the lifecycle meta argument in the database resource block and configure the destroy argument to false

Explanation

https://www.terraform.io/docs/configuration/resources.html under lifecycle

Question 40: Skipped
You are planning to use Terraform to manage resources in AWS. You opted to use environment variables to configure your AWS credentials access key as that would be more secure. Which Terraform environment variable would you use to accomplish that?

please select the correct answer:

- TF_VAR_AWS_ACCESS_KEY_ID="AKSALKDJKADKAJDA"
(Correct)
- export VAR_AWS_ACCESS_KEY_ID="AKSALKDJKADKAJDA"
- TF_VARS_AWS_ACCESS_KEY_ID="AKSALKDJKADKAJDA"
- TF_INPUT_AWS_ACCESS_KEY_ID="AKSALKDJKADKAJDA"

Explanation

https://www.terraform.io/docs/commands/environment-variables.html

prefix TF_VAR_ to any env variable. please remember that env variables have to be in capital letters and last as long as the shell.

Question 41: Skipped
When performing a terraform apply, Terraform outputs values are outputted to the CLI before any resources are created or updated.

True or False?

- False
(Correct)
- True

Explanation

always at the end of successful terraform apply once it knows the resource are created as it need to know the value of all attributes for created resources.

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

Question 42: Skipped

Which type of bracket does Terraform uses to constrain the code in a block?

please select the correct answer:

- {}
  (Correct)
- []
- /* */
- # #

Question 43: Skipped
You are planning to migrate your local backend to a standard backend. Please chose the supported standard backends supported by Terraform.

Please chose 4 correct answers:

- S3
  (Correct)
- GCS
  (Correct)
- Artifactory
  (Correct)
- AzureRM
  (Correct)
- AWSCodeCommit
- DynamoDB
- Terraform cloud

Explanation

https://www.terraform.io/docs/backends/types/index.html

Question 44: Skipped
You want to install an nginx server on your Google compute instance that you are going to create using a resource block with Terraform with all required arguments. Which would be the correct argument to add the script to the resource block?

please select correct answer:

-
```
resource "google_compute_instance" "VM_test" {
name = "VM-test"
...
script_data = <<-EOF
sudo apt-get install nginx
EOF
...
}
```
-
```
resource "google_compute_instance" "VM_test" {
name = "VM-test"
...
user-data = <<-EOF
sudo apt-get install nginx
EOF
...
}
```
-
```
resource "google_compute_instance" "VM_test" {
name = "VM-test"
...
metadata_startup_script = <<-EOF
sudo apt-get install nginx
EOF
...
}
```
(Correct)
-
```
resource "google_compute_instance" "VM_test" {
name = "VM-test"
...
user_data = <-EOF
sudo apt-get install nginx
EOF

}
```

Explanation

https://www.terraform.io/docs/providers/google/r/compute_instance.html

Question 45: Skipped
You have been requested to taint a resource. what would be the default action in the execution plan for this resource?

please select the correct answer:

- update in place
- destroy and create

(Correct)

- update and create
- create and destroy

Question 46: Skipped

Your terraform admin found out that Terraform supports parallelism. How many resources can be created by default in parallel by Terraform?

please select the correct answer:

- 40
- 10
(Correct)
- 20
- 60

Explanation

-parallelism=n Limit the number of parallel resource operations.

Defaults to 10.

Question 47: Skipped

Your team has created five EC2 instances using the Terraform count meta argument. You need to find the machine instance type of the third EC2 instance. Which terraform resource address would you use to accomplish that with the following configuration?

```
resource "aws_instance" "ec2" {
count = 5

ami = data.aws_ami.ami.id
instance_type = var.type
availability_zone = "us-east-2b"

}
```

please select the correct answer:

- aws_instance.ec2.instance_type[2]
- resource.aws_instance.ec2.instance_type[3]
- aws_instance.ec2[2].instance_type
(Correct)
- resource.aws_instance.ec2[3].instance_type
- aws_instance.ec2.instance_type[3]

Explanation

index is zero based number so third instance is index number 2.

resource_type.local_name[index].attribute

please refer to the count lecture 51 at 5min21

Question 48: Skipped

You have decided to use terraform to manage resources in Google cloud. What would be the required step to do that?

please select the correct answer:

- Configure google as a provider and Terraform will automatically create a .terraform directory and download the google binary.
- Configure google as a provider and Terraform will automatically create a .terraform/ directory and download the google binary when you initialize your Terraform root module
(Correct)
- Create a plugins subdirectory in the .terraform directory and configure the google provider. Terraform will download automatically the google plugin binary from the Terraform registry after initializing your Terraform root module
- Create a plugins subdirectory in the .terraform directory and download the google plugin binary from the Terraform registry. Configuring google as a provider will allow you to start deploying resources in Google cloud.

Question 49: Skipped

You have trying to use the for_each argument in your configuration in line with the Do Not Repeat Yourself developer principle. What types of value does the for_each argument support?

please select all correct answers:

- object
- map
(Correct)
- set
(Correct)

- list
- tuple

Explanation