Terraform Questions and Answers

Terraform 101

▶ **What is Terraform?**

Terraform: "HashiCorp Terraform is an infrastructure as code tool that lets you define both cloud and on-prem resources in human-readable configuration files that you can version, reuse, and share. You can then use a consistent workflow to provision and manage all of your infrastructure throughout its lifecycle. Terraform can manage low-level components like compute, storage, and networking resources, as well as high-level components like DNS entries and SaaS features."

▶ Details
What are the advantages in using Terraform or IaC in general?

- Full automation: In the past, resource creation, modification and removal were handled manually or by using a set of tooling. With Terraform or other IaC technologies, you manage the full lifecycle in an automated fashion.
- Modular and Reusable: Code that you write for certain purposes can be used and assembled in different ways. You can write code to create resources on a public cloud and it can be shared with other teams who can also use it in their account on the same (or different) cloud.
- Improved testing: Concepts like CI can be easily applied on IaC based projects and code snippets. This allow you to test and verify operations beforehand

▶ **What are some of Terraform features?**

- Declarative: Terraform uses the declarative approach (rather than the procedural one) in order to define end-status of the resources
- No agents: as opposed to other technologies (e.g. Puppet) where you use a model of agent and server, with Terraform you use the different APIs (of clouds, services, etc.) to perform the operations
- Community: Terraform has strong community who constantly publishes modules and fixes when needed. This ensures there is good modules maintenance and users can get support quite quickly at any point

▶ **What language does Terraform uses?**

A DSL called "HCL" (Hashicorp Configuration Language). A declarative language for defining infrastructure.

▶ **What's a typical Terraform workflow?**

1. Write Terraform definitions: `.tf` files written in HCL that described the desired infrastructure state (and run `terraform init` at the very beginning)
2. Review: With command such as `terraform plan` you can get a glance at what Terraform will perform with the written definitions
3. Apply definitions: With the command `terraform apply` Terraform will apply the given definitions, by adding, modifying or removing the resources

This is a manual process. Most of the time this is automated so user submits a PR/MR to propose terraform changes, there is a process to test these changes and once merged they are applied (`terraform apply`).

▶ **What are some use cases for using Terraform?**

- Infra provisioning and management: You need to automated or code your infra so you are able to test it easily, apply it and make any changes necessary.
- Multi-cloud environment: You manage infrastructure on different clouds, but looking for a consistent way to do it across the clouds
- Consistent environments: You manage environments such as test, production, staging, ... and looking for a way to have them consistent so any modification in one of them, applies to other environments as well

▶ **What's the difference between Terraform and technologies such as Ansible, Puppet, Chef, etc.**

Terraform is considered to be an IaC technology. It's used for provisioning resources, for managing infrastructure on different platforms.

Ansible, Puppet and Chef are Configuration Management technologies. They are used once there is an instance running and you would like to apply some configuration on it like installing an application, applying security policy, etc.

To be clear, CM tools can be used to provision resources so in the end goal of having infrastructure, both Terraform and something like Ansible, can achieve the same result. The difference is in the how. Ansible doesn't saves the state of resources, it doesn't know how many instances there are in your environment as opposed to Terraform. At the same time while Terraform can perform configuration management tasks, it has less modules support for that specific goal and it doesn't track the task execution state as Ansible. The differences are there and it's most of the time recommended to mix the technologies, so Terraform used for managing infrastructure and CM technologies used for configuration on top of that infrastructure.

Terraform Hands-On Basics

▶ **Explain the following block of Terraform code**

```
resource "aws_instance" "some-instance" {
  ami           = "ami-201720221991yay"
  instance_type = "t2.micro
}
```

It's a resource of type "aws_instance" used to provision an instance. The name of the resource (NOT INSTANCE) is "some-instance".

The instance itself will be provisioned with type "t2.micro" and using an image of the AMI "ami-201720221991yay".

▶ **What do you do next after writing the following in main.tf file?**

```
```

```
resource "aws_instance" "some-instance" {
  ami           = "ami-201720221991yay"
  instance_type = "t2.micro
}
```

Run `terraform init`. This will scan the code in the directory to figure out which providers are used (in this case AWS provider) and will download them.

▶ **You've executed `terraform init` and now you would like to move forward to creating the resources but you have concerns and would like to make be 100% sure on what you are going to execute. What should you be doing?**

Execute `terraform plan`. That will provide a detailed information on what Terraform will do once you apply the changes.

▶ **You've downloaded the providers, seen the what Terraform will do (with terraform plan) and you are ready to actually apply the changes. What should you do next?**

Run `terraform apply`. That will apply the changes described in your .tf files.

▶ **Explain the meaning of the following strings that seen at the beginning of each line When you run `terraform apply` - '+' - '-' - '-/+'**

- '+' - The resource or attribute is going to be added
- '-' - the resource or attribute is going to be removed
- '-/+' - the resource or attribute is going to be replaced

▶ **How to cleanup Terraform resources? Why the user shold be careful doing so?**

`terraform destroy` will cleanup all the resources tracked by Terraform.

A user should be careful with this command because there is no way to revert it. Sure, you can always run again "apply" but that can take time, generates completely new resources, etc.

Dependencies

▶ Details
Sometimes you need to reference some resources in the same or separate .tf file. Why and how it's done?

Why: because resources are sometimes connected or need to be connected. For example, you create a AWS VPC with **aws_vpc** resource but, at the same time you want to create and attach IGW to it. For that you'll create a **aws_internet_gateway** resource and then and in this resource you will refer **aws_vpc** resource.

How:

Using the syntax ".."

In your AWS instance it would like that:

```
```

```
resource "aws_vpc" "terraform_test_vpc" {
  cidr_block           = var.vpc_cidr
  enable_dns_hostnames = true
  enable_dns_support   = true

  tags = {
    Name = var.vpc_tag_name
  }
}

resource "aws_internet_gateway" "terraform_test_internet_gateway" {
  vpc_id = aws_vpc.terraform_test_vpc.id

  tags = {
    Name = "terraform_test_internet_gateway"
  }
}
```

▶ **Does it matter in which order Terraform creates resources?**

Yes, when there is a dependency between different Terraform resources, you want the resources to be created in the right order and this is exactly what Terraform does.

To make it ever more clear, if you have a resource X that references the ID of resource Y, it doesn't makes sense to create first resource X because it won't have any ID to get from a resource that wasn't created yet.

▶ **Is there a way to print/see the dependencies between the different resources?**

Yes, with `terraform graph`

The output is in DOT - A graph description language.

Providers

▶ **Explain what is a "provider"**

terraform.io: "Terraform relies on plugins called "providers" to interact with cloud providers, SaaS providers, and other APIs...Each provider adds a set of resource types and/or data sources that Terraform can manage. Every resource type is implemented by a provider; without providers, Terraform can't manage any kind of infrastructure."

▶ **Where can you find publicly available providers?**

In the Terraform Registry

▶ **What are the names of the providers in this case?**

```
terraform {
    required_providers {
      aws = {
        source  = "hashicorp/aws"
      }
      azurerm = {
        source  = "hashicorp/azurerm"
        version = "~> 3.0.2"
      }
    }
  }
```

azurerm and aws

▶ **How to install a provider?**

You write a provider block like the following one and run `terraform init`

```
provider "aws" {
  region = "us-west-1"
}
```

▶ **True or False? Applying the following Terraform configuration will fail since no source or version specific for 'aws' provider**

```
terraform {
    required_providers {
      aws = {}
    }
  }
```

False. It will look for "aws" provider in the public Terraform registry and will take the latest version.

▶ **Write a configuration of a Terraform provider (any type you would like)**

AWS is one of the most popular providers in Terraform. Here is an example of how to configure it to use one specific region and specifying a specific version of the provider

```
terraform {
  required_providers {
    aws = {
      source  = "hashicorp/aws"
      version = "~> 3.0"
    }
  }
}

# Configure the AWS Provider
provider "aws" {
  region = "us-west-2"
}
```

▶ **Where Terraform installs providers from by default?**

By default Terraform providers are installed from Terraform Registry

▶ **What is the Terraform Registry?**

The Terraform Registry provides a centralized location for official and community-managed providers and modules.

▶ **Where providers are downloaded to? (when for example you run `terraform init`)**

`.terraform` directory.

▶ **Describe in high level what happens behind the scenes when you run terraform init on on the following Terraform configuration**

```
terraform {
  required_providers {
    aws = {
      source  = "hashicorp/aws"
      version = "~> 3.0"
    }
  }
}
```

1. Terraform checks if there is an aws provider in this address: `registry.terraform.io/hashicorp/aws`
2. Installs latest version of aws provider (assuming the URL exists and valid)

▶ **True or False? You can install providers only from hashicorp**

False. You can specify any provider from any URL, not only those from hashicorp.

Variables

Input Variables

▶ **What input variables are good for in Terraform?**

Variables allow you define piece of data in one location instead of repeating the hardcoded value of it in multiple different locations. Then when you need to modify the variable's value, you do it in one location instead of changing each one of the hardcoded values.

▶ **What type of input variables are supported in Terraform?**

```
string
number
```

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

▶ **What's the default input variable type in Terraform?**

`any`

▶ **What ways are there to pass values for input variables?**

- Using `-var` option in the CLI
- Using a file by using the `-var-file` option in the CLI
- Environment variable that starts with `TF_VAR_<VAR_NAME>`

If no value given, user will be prompted to provide one.

▶ **How to reference a variable?**

Using the syntax `var.<VAR_NAME>`

▶ **What is the effect of setting variable as "sensitive"?**

It doesn't show its value when you run `terraform apply` or `terraform plan` but eventually it's still recorded in the state file.

▶ **True or False? If an expression's result depends on a sensitive variable, it will be treated as sensitive as well**

True

▶ **The same variable is defined in the following places:**

- The file `terraform.tfvars`
- Environment variable
- Using `-var` or `-var-file`

According to variable precedence, which source will be used first?

The order is:

- Environment variable
- The file `terraform.tfvars`
- Using `-var` or `-var-file`

▶ **Whenever you run terraform apply, it prompts to enter a value for a given variable. How to avoid being prompted?**

While removing the variable is theoretically a correct answer, it will probably fail the execution.

You can use something like the `-var` option to provide the value and avoid being prompted to insert a value. Another option is to run `export TF_VAR_<VAR_NAME>=<VALUE>`.

Output Variables

▶ **What are output variables? Why do we need them?**

Output variable allow you to display/print certain piece of data as part of Terraform execution.

The most common use case for it is probably to print the IP address of an instance. Imagine you provision an instance and you would like to know what the IP address to connect to it. Instead of looking for it for the console/OS, you can use the output variable and print that piece of information to the screen

▶ **Explain the "sensitive" parameter of output variable**

When set to "true", Terraform will avoid logging output variable's data. The use case for it is sensitive data such as password or private keys.

▶ **Explain the "depends" parameter of output variable**

It is used to set explicitly dependency between the output variable and any other resource. Use case: some piece of information is available only once another resource is ready.

Locals

▶ **What are locals?**

Similarly to variables they serve as placeholders for data and values. Differently from variables, users can't override them by passing different values.

▶ **What's the use case for using locals?**

You have multiple hardcoded values that repeat themselves in different sections, but at the same time you don't want to expose them as in, allow users to override values.

Variables Hands-On

▶ **Demonstrate input variable definition with type, description and default parameters**

```
variable "app_id" {
  type = string
  description = "The id of application"
  default = "some_value"
}
```

Unrelated note: variables are usually defined in their own file (vars.tf for example).

▶ **How to define an input variable which is an object with attributes "model" (string), "color" (string), year (number)?**

```
variable "car_model" {
  description = "Car model object"
  type        = object({
    model    = string
    color    = string
    year     = number
  })
}
```

Note: you can also define a default for it.

▶ **How to reference variables?**

Variable are referenced with `var.VARIABLE_NAME` syntax. Let's have a look at an example:

vars.tf:

```
variable "memory" {
  type = string
  default "8192"
}

variable "cpu" {
  type = string
  default = "4"
}
```

main.tf:

```
resource "libvirt_domain" "vm1" {
   name = "vm1"
   memory = var.memory
   cpu = var.cpu
}
```

▶ **How to reference variable from inside of string literal? (bonus question: how that type of expression is called?)**

Using the syntax: `"${var.VAR_NAME}"`. It's called "interpolation".

Very common to see it used in user_data attribute related to instances.

```
user_data = <<-EOF
            This is some fabulos string
            It demonstrates how to use interpolation
            Yes, it's truly ${var.awesome_or_meh}
            EOF
```

▶ **How can list all outputs without applying Terraform changes?**

`terraform output` will list all outputs without applying any changes

▶ **Can you see the output of specific variable without applying terrafom changes?**

Yes, with `terraform output <OUTPUT_VAR>`.

Very useful for scripts 😃

▶ **Demonstrate how to define locals**

```
locals {
  x = 2
  y = "o"
  z = 2.2
}
```

▶ **Demonstrate how to use a local**

if we defined something like this

```
locals {
  x = 2
}
```

then to use it, you have to use something like this: `local.x`

Data Sources

▶ **Explain data sources in Terraform**

- Data sources used to get data from providers or in general from external resources to Terraform (e.g. public clouds like AWS, GCP, Azure).
- Data sources used for reading. They are not modifying or creating anything
- Many providers expose multiple data sources

▶ **Demonstrate how to use data sources**

```
data "aws_vpc" "default {
  default = true
}
```

▶ **How to get data out of a data source?**

The general syntax is `data.<PROVIDER_AND_TYPE>.<NAME>.<ATTRBIUTE>`

So if you defined the following data source

```
data "aws_vpc" "default {
  default = true
}
```

You can retrieve the ID attribute this way: `data.aws_vpc.default.id`

▶ **Is there such a thing as combining data sources? What would be the use case?**

Yes, you can define a data source while using another data source as a filter for example.

Let's say we want to get AWS subnets but only from our default VPC:

```
data "aws_subnets" "default" {
  filter {
    name   = "vpc-id"
    values = [data.aws_vpc.default.id]
  }
}
```

Lifecycle

▶ **When you update a resource, how it works?**

By default the current resource is deleted, a new one is created and any references pointing the old resource are updated to point the new resource

▶ **Is it possible to modify the default lifecycle? How? Why?**

Yes, it's possible. There are different lifecycles one can choose from. For example "create_before_destroy" which inverts the order and first creates the new resource, updates all the references from old resource to the new resource and then removes the old resource.

How to use it:

```
lifecycle {
  create_before_destroy = true
}
```

Why to use it in the first place: you might have resources that have dependency where they dependency itself is immutable (= you can't modify it hence you have to create a new one), in such case the default lifecycle won't work because you won't be able to remove the resource that has the dependency as it still references an old resource. AWS ASG + launch configurations is a good example of such use case.

▶ **You've deployed a virtual machine with Terraform and you would like to pass data to it (or execute some commands). Which concept of Terraform would you use?**

Provisioners

Provisioners

▶ **What are "Provisioners"? What they are used for?**

Provisioners can be described as plugin to use with Terraform, usually focusing on the aspect of service configuration and make it operational.

Few example of provisioners:

- Run configuration management on a provisioned instance using technology like Ansible, Chef or Puppet.
- Copying files
- Executing remote scripts

▶ **Why is it often recommended to use provisioners as last resort?**

Since a provisioner can run a variety of actions, it's not always feasible to plan and understand what will happen when running a certain provisioner. For this reason, it's usually recommended to use Terraform built-in option, whenever's possible.

▶ **What is `local-exec` and `remote-exec` in the context of provisioners?**
▶ **What is a "tainted resource"?**

It's a resource which was successfully created but failed during provisioning. Terraform will fail and mark this resource as "tainted".

▶ **What `terraform taint` does?**
▶ **What is a data source? In what scenarios for example would need to use it?**

There are quite a few cases you might need to use them:

- you want to reference resources not managed through terraform
- you want to reference resources managed by a different terraform module
- you want to cleanly compute a value with typechecking, such as with `aws_iam_policy_document`

▶ **What are output variables and what `terraform output` does?**
▶ **Explain `remote-exec` and `local-exec`**
▶ **Explain "Remote State". When would you use it and how?**
▶ **Explain "State Locking"**
▶ **Aside from `.tfvars` files or CLI arguments, how can you inject dependencies from other modules?**
▶ **How do you import existing resource using Terraform import?**

1. Identify which resource you want to import.
2. Write terraform code matching configuration of that resource.
3. Run terraform command `terraform import RESOURCE ID`

eg. Let's say you want to import an aws instance. Then you'll perform following:

1. Identify that aws instance in console
2. Refer to it's configuration and write Terraform code which will look something like:

```
resource "aws_instance" "tf_aws_instance" {
  ami           = data.aws_ami.ubuntu.id
  instance_type = "t3.micro"

  tags = {
    Name = "import-me"
  }
}
```

3. Run terraform command `terraform import aws_instance.tf_aws_instance i-12345678`

State

▶ **What's Terraform State?**

terraform.io: "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."

In other words, it's a mechanism in Terraform to track resources you've created or cleaned up. This is how terraform knows what to update/create/delete when you run `terraform apply` and also other commands like `terraform destroy`.

▶ **Where Terraform state is stored?**

There is more than one answer to this question. It's very much depends on whether you share it with others or it's only local in your Terraform directory, but taking a beginner's case, when you run terraform in a directory, the state will be stored in that directory in `terraform.tfstate` file.

▶ **Can you name three different things included in the state file?**

- The representation of resources - JSON format of the resources, their attributes, IDs, ... everything that required to identify the resource and also anything that was included in the .tf files on these resources
- Terraform version
- Outputs

▶ **Why does it matter where you store the tfstate file? In your answer make sure to address the following:**

- Public vs. Private

- Git repository vs. Other locations

- tfstate contains credentials in plain text. You don't want to put it in publicly shared location

- tfstate shouldn't be modified concurrently so putting it in a shared location available for everyone with "write" permissions might lead to issues. (Terraform remote state doesn't has this problem).

- tfstate is an important file. As such, it might be better to put it in a location that has regular backups and good security.

As such, tfstate shouldn't be stored in git repositories. secured storage such as secured buckets, is a better option.

▶ **True or False? it's common to edit terraform state file directly by hand and even recommended for many different use cases**

False. You should avoid as much possible to edit Terraform state files directly by hand.

▶ **Why storing state file locally on your computer may be problematic?**

In general, storing state file on your computer isn't a problem. It starts to be a problem when you are part of a team that uses Terraform and then you would like to make sure it's shared. In addition to being shared, you want to be able to handle the fact that different teams members can edit the file and can do it at the same time, so locking is quite an important aspect as well.

▶ **Mention some best practices related to tfstate**

- Don't edit it manually. tfstate was designed to be manipulated by terraform and not by users directly.
- Store it in secured location (since it can include credentials and sensitive data in general)
- Backup it regularly so you can roll-back easily when needed
- Store it in remote shared storage. This is especially needed when working in a team and the state can be updated by any of the team members
- Enabled versioning if the storage where you store the state file, supports it. Versioning is great for backups and roll-backs in case of an issue.

▶ **How and why concurrent edits of the state file should be avoided?**

If there are two users or processes concurrently editing the state file it can result in invalid state file that doesn't actually represents the state of resources.

To avoid that, Terraform can apply state locking if the backend supports that. For example, AWS s3 supports state locking and consistency via DynamoDB. Often, if the backend supports it, Terraform will make use of state locking automatically so nothing is required from the user to activate it.

▶ **Describe how you manage state file(s) when you have multiple environments (e.g. development, staging and production)**

Probably no right or wrong answer here, but it seems, based on different source, that the overall preferred way is to have a dedicated state file per environment.

▶ **Why storing the state in versioned control repo is not a good idea?**

- Sensitive data: some resources may specify sensitive data (like passwords and tokens) and everything in a state file is stored in plain text
- Prone to errors: when working with Git repos, you mayo often find yourself switch branches, checkout specific commits, perform rebases, ... all these operations may end up in you eventually performing `terraform apply` on non-latest version of your Terraform code

Terraform Backend

▶ **What's a Terraform backend? What is the default backend?**

Terraform backend determines how the Terraform state is stored and loaded. By default the state is local, but it's possible to set a remote backend

▶ **Describe how to set a remote backend of any type you choose**

Let's say we chose use Amazon s3 as a remote Terraform backend where we can store Terraform's state.

1. Write Terraform code for creating an s3 bucket
    1. It would be a good idea to add lifecycle of "prevent_destroy" to it so it's not accidentally deleted
2. Enable versioning (add a resource of "aws_s3_bucket_versioning")
3. Encrypt the bucket ("aws_s3_bucket_server_side_encryption_configuration")
4. Block public access
5. Handle locking. One way is to add DB for it
6. Add the point you'll want to run init and apply commands to avoid an issue where you at the same time create the resources for remote backend and also switch to a remote backend
7. Once resources were created, add Terraform backend code

```
terraform {
  backend "s3" {
    bucket ...
  }
}
```

7. Run `teraform init` as it will configure the backend

▶ **How `terraform apply` workflow is different when a remote backend is used?**

It starts with acquiring a state lock so others can't modify the state at the same time.

▶ **What would be te process of switching back from remote backend to local?**

1. You remove the backend code and perform `terraform init` to switch back to `local` backend
2. You remove the resources that are the remote backend itself

▶ **True or False? it's NOT possible to use variable in a backend configuration**

That's true and quite a limitation as it means you'll have to go to the resources of the remote backend and copy some values to the backend configuration.

One way to deal with it is using partial configurations in a completely separate file from the backend itself and then load them with `terraform init -backend-config=some_backend_partial_conf.hcl`

▶ **Is there a way to obtain information from a remote backend/state usign Terraform?**

Yes, using the concept of data sources. There is a data source for a remote state called "terraform_remote_state".

You can use it the following syntax `data.terraform_remote_state.<NAME>.outputs.<ATTRIBUTE>`

Workspaces

▶ **Explain what is a Terraform workspace**

developer.hashicorp.com: "The persistent data stored in the backend belongs to a workspace. The backend initially has only one workspace containing one Terraform state associated with that configuration. Some backends support multiple named workspaces, allowing multiple states to be associated with a single configuration."

▶ **True or False? Each workspace has its own state file**

True

▶ **Why workspaces might not be the best solution for managing states for different environemnts? like staging and production**

One reason is that all the workspaces are stored in one location (as in one backend) and usually you don't want to use the same access control and authentication for both staging and production for obvious reasons. Also working in workspaces is quite prone to human errors as you might accidently think you are in one workspace, while you are working a completely different one.

State Hands-On

▶ **Which command will produce a state file?**

`terraform apply`

▶ **How to inspect current state?**

`terraform show`

▶ **How to list resources created with Terraform?**

`terraform state list`

▶ **How do you rename an existing resource?**

`terraform state mv`

▶ **How to create a new workspace?**

`terraform workspace new <WORKSPACE_NAME>`

▶ **How to identify which workspace are you using?**

`terraform workspace show`

Terraform Structures and Syntax

Lists

▶ **How to define an input variable which is a list of numbers?**

```
variable "list_of_nums" {
  type = list(number)
  description = "An example of list of numbers"
  default = [2, 0, 1, 7]
}
```

▶ **How to create a number of resources based on the length of a list?**

```
resource "some_resource" "some_name" {
  count = length(var.some_list)
}
```

▶ **You have a list variable called "users". How to access the second item in that list and attribute called "name"?**

`users[1].name`

▶ **You have a list variable called "users". How to access attribute "name" of all items?**

`users[*].name`

Loops

▶ **What loops are useful for in Terraform?**

The most common use case is when you need to create multiple resources with only a slight difference (like different name). Instead of defining multiple separate resources, you can define it once and create multiple instances of it using loops.

▶ **Demonstrate how to define a simple Terraform loop**

```
resource "aws_instance" "server" {
  count = 15
}
```

The above configuration will create 15 aws instances.

▶ **How to create multiple AWS instances but each with a different name?**

```
resource "aws_instance" "server" {
  count = 6

  tags = {
    Name = "instance-${count.index}"
  }
}
```

The above configuration will create 6 instances, each with a different name.

▶ **You have the following variable defined in Terraform**

```
variable "users" {
  type    = list(string)
  default = ["mario", "luigi", "peach"]
}
```

How to use it to create users on one of the public clouds (or any other platform, infra)?

```
resource "aws_iam_user" "user" {
  count = length(var.users)

  name = var.users[count.index]
}
```

▶ **Is there any limitation to "count" meta-argument?**

- `count` isn't supported within an inline block
- It's quite limited when it comes to lists.You'll notice that modifying items in lists or even operations like removal sometimes interpreted in a way you didn't expect. For example, removing an item from a list, may shift other items to a new position and since each position represents a resource with count, that may lead to a result where wrong resources are being modified and removed. There are ways to do deal it, but still using count with lists is not always straightforward

▶ **What's a for_each loop? How is it different from "count"?**

- for_each can applied only on collections like maps or sets (as opposed to count that can be applied on lists)
- for_each helps to deal with the limitation of `count` which isn't optimal for use cases of modifying lists
- for_each supports inline blocks as opposed to `count`

▶ **Demonstrate how to use the for_each loop**

```
resource “google_compute_instance” “instances” {

  for_each = var.names_map
  name = each.value
}
```

▶ **The following resource tries to use for_each loop on a list of string but it fails, why?**

```
resource “google_compute_instance” “instances” {

  for_each = var.names
  name = each.value
}
```

for_each can applied only on collections like maps or sets so the list should be converted to a set like this: `for_each = toset(var.names)`

▶ **How to use for_each loop for inline blocks?**

```
resouce "some_instance" "instance" {

dynamic "tag" {
  for_each = var.tags

  content {
    key   = tag.key
    value = tag.value
    }
  }
}
```

▶ **There is a list variable called "users". You would like to define an output variable with a value of all users in uppercase. How to achieve that?**

```
output "users" {
  value = [for name in var.user_names : upper(name)]
}
```

▶ **What's the result of the following code?**

```
resource "random_integer" "num" {
  min = 20
  max = 17
}

resource "aws_instance" "instances" {
  count = random_integer.num.results
}
```

The above code will fail as it's not possible to reference resource outputs with count, because Terraform has to compute count before any resources are created (or modified).

▶ **There is a variable called "values" with the following value: ["mario", "luigi", "peach"]. How to create an output variable with the string value of the items in the list: "mario, luigi, peach," ?**

```
output "users" {
  value = "%{ for name in var.values }${name}, %{ endfor }"
}
```

▶ **There is a list variable called "users". You would like to define an output variable with a value of all users in uppercase but only if the name is longer than 3 characters. How to achieve that?**

```
output "users" {
  value = [for name in var.user_names : upper(name) if length(name) > 3]
}
```

Maps

▶ **There is a map called "instances"**

- How to extract only the values of that map?
- How to extract only the attribute "name" from each value?
- Using the values built-in function: `values(instances)`
- `values(instances)[*].name`

▶ **You have a map variable, called "users", with the keys "name" and "age". Define an output list variable with the following "my name is {name} and my age is {age}"**

```
output "name_and_age" {
  value = [for name, age in var.users : "my name is ${name} and my age is ${age}"]
}
```

▶ **You have a map variable, called "users", with the keys "name" (string) and "age" (float). Define an output map variable with the key being name in uppercase and value being age in the closest whole number**

```
output "name_and_age" {
  value = {for name, age in var.users : upper(name) => floor(age)
}
```

Conditionals

▶ **How to use conditional expressions in Terraform?**

`some_condition ? "value_if_true" : "value_if_false"`

▶ **Explain the following condition:** `var.x ? 1 : 0`

If `x` evaluated to true, the result is 1, otherwise (if false) the result is 0.

▶ **Explain the following condition:** `var.x != "" ? var.x : "yay"`

If `x` is an empty string the result is "yay", otherwise it's the value of `x` variable

▶ **Can conditionals be used with meta-arguments?**

Yes, for example the "count" meta-argument:

```
resource "aws_instance" "server" {
  count = var.amount ? 1 : 0
  ...
}
```

▶ **Is it possible to combine conditionals and loop?**

Yes, for example:

```
dynamic "tag" {
  for_each = {
    for key, value in var.tags:
    key => value
    if key != ""
  }
}
```

Misc

▶ **What are meta-arguments in Terraform?**

Arguments that affect the lifecycle of a resources (its creation, modification, ...) and supported by Terraform regardless to the type of resource in which they are used.

Some examples:

- count: how many resources to create out of one definition of a resource
- lifecycle: how to treat resource creation or removal

▶ **What meta-arguments are you familiar with?**

- count: how many resources to create out of one definition of a resource
- lifecycle: how to treat resource creation or removal
- depends_on: create a dependency between resources

▶ **What `templatefile` function does?**

Renders a template file and returns the result as string.

▶ **You are trying to use templatefile as part of a module and you use a relative path to load a file but sometimes it fails, especially when others try to reuse the module. How can you deal with that?**

Switch relative paths with what is known as path references. These are fixes paths like module root path, module expression file path, etc.

▶ **How do you test terraform syntax?**

A valid answer could be "I write Terraform configuration and try to execute it" but this makes testing cumbersome and quite complex in general.

There is `terraform console` command which allows you to easily execute terraform functions and experiment with general syntax.

▶ **True or False? Terraform console should be used carefully as it may modify your resources**

False. terraform console is ready-only.

▶ **You need to render a template and get it as string. Which function would you use?**

`templatefile` function.

▶ **Explain what `depends_on` used for and given an example**

`depends_on` used to create a dependency between resources in Terraform. For example, there is an application you would like to deploy in a cluster. If the cluster isn't ready (and also managed by Terraform of course) then you can't deploy the app. In this case, you will define "depends_on" in the app configuration and its value will be the cluster resource.

Modules

▶ **Explain Modules**

Terraform.io: "A module is a container for multiple resources that are used together. Modules can be used to create lightweight abstractions, so that you can describe your infrastructure in terms of its architecture, rather than directly in terms of physical objects."

In addition, modules are great for creating reusable Terraform code that can be shared and used not only between different repositories but even within the same repo, between different environments (like staging and production).

▶ **What makes a Terraform code module? In other words, what a module is from practical perspective?**

Basically any file or files in a directory is a module in Terraform. There is no special syntax to use in order to define a module.

▶ **How do you test a Terraform module?**

There are multiple answers, but the most common answer would likely to be using the tool `terratest`, and to test that a module can be initialized, can create resources, and can destroy those resources cleanly.

▶ **When creating a module, do you prefer to use inline blocks, separate resources or both? why?**

No right or wrong here.

Personally, I prefer to use only separate resources in modules as it makes modules more flexible. So if a resource includes inline blocks, that may limit you at some point.

▶ **True or False? Module source can be only local path**

False. It can be a Git URL, HTTP URL, ... for example:

```
module "some_module" {

  source = "github.com/foo/modules/bar?ref=v0.1"
}
```

▶ **Where can you obtain Terraform modules?**

Terraform modules can be found at the Terrafrom registry

▶ **You noticed there are relative paths in some of your modules and you would like to change that. What can you do and why is that a problem in the first place?**

Relative paths usually work fine in your own environment as you are familiar with the layout and paths used, but when sharing a module and making it reusable, you may bump into issues as it runs on different environments where the relative paths may no longer be relevant.

A better approach would be to use `path reference` like one of the following:

- `path.module`: the path of the module where the expression is used
- `path.cwd`: the path of the current working directory
- `path.root`: the path of the root module

Modules Hands-On

▶ **How to use a module?**

The general syntax is:

```
module "<MODULE_NAME>" {
  source = "<MODULE_SOURCE>"

  ...
}
```

The critical part is the source which you use to tell Terraform where the module can be found.

▶ **Demonstrate using a module called "amazing_modle" in the path "../modules/amazing-module"**

```
module "amazing_module" {
  source = "../modules/amazing-module"
}
```

▶ **What should be done every time you modify the source parameter of a module?**

`terraform init` should be executed as it takes care of downloading and installing the module from the new path.

▶ **How to access module output variables?**

the general syntax is `module.<MODULE_NAME>.<OUTPUT_VAR_NAME>`

▶ **You would like to load and render a file from module directory. How to do that?**

script = templatesfile("${path.module}/user-data.sh", { ... })

▶ **There is a module to create a compute instance. How would you use the module to create three separate instances?**

starting with Terraform 0.13, the `count` meta-argument can be used with modules. So you could use something like this:

```
module "instances" {
  source = "/some/module/path"

  count = 3
}
```

You can also use it in outputs vars: `value = module.instances[*]`

Import

▶ **Explain Terraform's import functionality**

`terraform import` is a CLI command used for importing an existing infrastructure into Terraform's state.

It's does NOT create the definitions/configuration for creating such infrastructure.

▶ **State two use cases where you would use `terraform import`**

1. You have existing resources in one of the providers and they are not managed by Terraform (as in not included in the state)
2. You lost your tfstate file and need to rebuild it

Version Control

▶ **You have a Git repository with Terraform files but no .gitignore. What would you add to a .gitignore file in Terraform repository?**

```
.terraform
*.tfstate
*.tfstate.backup
```

You don't want to store state file nor any downloaded providers in .terraform directory. It also doesn't makes sense to share/store the state backup files.

AWS

▶ **What happens if you update user_data in the following case apply the changes?**

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

user_data = <<-EOF
            #!/bin/bash
            echo "Hello, World" > index.xhtml
            EOF
```

Nothing, because user_data is executed on boot so if an instance is already running, it won't change anything.

To make it effective you'll have to use `user_data_replace_on_change = true`.

▶ **You manage ASG with Terraform which means you also have "aws_launch_configuration" resources. The problem is that launch configurations are immutable and sometimes you need to change them. This creates a problem where Terraform isn't able to delete ASG because they reference old launch configuration. How to do deal with it?**

Add the following to "aws_launch_configuration" resource

```
lifecycle {
  create_before_destroy = true
}
```

This will change the order of how Terraform works. First it will create the new resource (launch configuration). then it will update other resources to reference the new launch configuration and finally, it will remove old resources

▶ **How to manage multiple regions in AWS provider configuration?**

```
provider "aws" {
  region = "us-west-1"
  alias = "west_region"
}
```

```
provider "aws" {
  region = "us-east-1"
  alias = "east_region"
}

data "aws_region" "west_region" {
  provider = aws.west_region
}

data "aws_region" "east_region" {
  provider = aws.east_region
}
```

To use it:

```
resource "aws_instance" "west_region_instance" {
  provider = aws.west_region
  instance_type = "t2.micro"
  ...
}
```

▶ **Assuming you have multiple regions configured and you would like to use a module in one of them. How to achieve that?**

```
module "some_module" {
  source = "..."

  providers = {
    aws = aws.some_region
  }

  ...
}
```

▶ **How to manage multiple AWS accounts?**

One way is to define multiple different provider blocks, each with its own "assume_role"

```
provider "aws" {
  region = "us-west-1"
  alias = "some-region"

  assume_role {
    role_arn = "arn:aws:iam::<SOME_ACCOUNT_ID>:role/<SOME_ROLE_NAME>"
  }
}
```

Validations

▶ **How would you enforce users that use your variables to provide values with certain constraints? For example, a number greater than 1**

Using `validation` block

```
variable "some_var" {
  type = number

  validation {
    condition = var.some_var > 1
    error_message = "you have to specify a number greater than 1"
  }

}
```

Secrets

▶ **What's the issue with the following provider configuration?**

```
provider "aws" {
  region = "us-west-1"

  access_key = "blipblopblap"
  secret_key = "bipbopbipbop"
}
```

It's not secure! you should never store credentials in plain text this way.

▶ **What can you do to NOT store provider credentials in Terraform configuration files in plain text?**

1. Use environment variables
2. Use password CLIs (like 1Password which is generic but there also specific provider options like aws-vault)

▶ **How can you manage secrets/credentials in CI/CD?**

That very much depends on the CI/CD system/platform you are using.

- GitHub Actions: Use Open ID Connect (OIDC) to establish connection with your provider. You then can specify in your GitHub Actions workflow the following:

```
- uses: aws-actions/configure-aws-credentials@v1
with:
 role-to-assume: arn:aws:iam::someIamRole
 aws-region: ...
```

- Jenkins: If Jenkins runs on the provider, you can use the provider access entities (like roles, policies, ...) to grant the instance, on which Jenkins is running, access control
- CircleCI: you can use `CircleCI Context` and then specify it in your CircleCI config file

```
context:
- some-context
```

▶ **What are the pros and cons of using environment variables for managing secrets in Terraform configurations?**

Pros:

- You avoid using secrets directly in configurations in plain text
- free (no need to pay for secret management platforms/solutions)
- Straightforward to use

Cons:

- Configurations might not be usable without the environment variables which may make impact the user experience as the user has to know what environment variables he should pass for everything to work properly
- Mostly managed outside of Terraform mechanisms which makes it hard to enforce, track, ... anything that is related to secrets when it depends on the user to pass environment variables

▶ **True or False? If you pass secrets with environment variables, they are not visible in your state file**

False. State files include sensitive data as it is. Which means it's very important that wherever you store your state file, it's encrypted and accessible only to those who should be able to access it.

▶ **True or False? If you pass secrets from a centralized secrets store (like Hashicorp Vault) they are not visible in plan files (terraform plan)**

False. It doesn't matter where your secrets store (file, environment variables, centralized secrets store), they will be visible in both state file and plan output.