Points To Ponder (**random but important**)

Nilesh Joshi

IaC matrix:

Name	Key Feature					
	Provisioning Tool	Easy To use	Free and OpenSource	Declarative	Cloud Agnostic	Expressive and Extendble
Ansible		X	X		X	X
Chef			X	X	X	X
Puppet			X	X	X	X
SlatStack		X	X	X	X	X
Terraform	X	X	X	X	X	X
Pulumi	X		X		X	X
AWS CloudFormation	X	X		X		
GCP Resource Manager	X	X		X		
Azure Resource Manager	X			X		

if `terraform plan` is running slow, turn off trace level logging and consider increasing parallelism (`-parallelism=n`)

WHEN MIGHT MY PLAN FAIL?

Terraform plans can fail for many reasons, such as if your configuration code is invalid, or if there's a versioning issue, or even network related problems. Sometimes, albeit rarely, the plan will fail as a result of a bug in the provider's source code. You need to carefully read whatever error message you receive to know for sure. For more verbose logs, you can turn on trace level logging by setting the environment variable TF_LOG to a non-zero value, e.g. `export TF_LOG=1`

WARNING

It's important to not edit, delete or otherwise tamper with the `terraform.tfstate` file, or else Terraform could potentially lose track of the resources it manages. It is possible to restore a corrupted or missing state file, but it's difficult and time consuming to do so.

- By running terraform plan, the current state is refreshed and the configuration is consulted to generate an action plan. The plan includes all actions to be taken: which resources will be created, destroyed or modified.

==========

- Using terraform graph, the plan can be visualized to show dependent ordering.

==========

- Terraform maintains a list of dependencies in the state file so that it can properly deal with dependencies that no longer exist in the current configuration.

==========

- Terraform state enables the mapping of real world instances to resources in a configuration, improved performance of the planning engine, and collaboration of teams.

==========

- By default Terraform will refresh the state before each planning run, but to improve performance Terraform can be told to skip the refresh with the --refresh=false argument. The -target argument can be used to specify a particular resource to refresh, without triggering a full refresh of the state. In these scenarios, the cached state is treated as the record of truth.

NOTE: Choosing not to refresh the state means that the reality of your infrastructure deployment may not match what is in the state file.

==========

- You can have multiple different providers or multiple instances of the same provider in a configuration. It's important to understand how to use the alias argument and the use cases for multiple provider instances.

==========

- Provisioners are a measure of last resort. Remote-exec will run a script on the remote machine through WinRM or ssh, and local-exec will run a script on your local machine.

==========

- Tainted Resources

If a resource is successfully created but fails a provisioning step, Terraform will error and mark the resource as tainted. A resource that is tainted still exists, but shouldn't be considered safe to use, since provisioning failed. When you generate your next execution plan, Terraform will remove any tainted resources and create new resources, attempting to provision them again after creation.

Terraform taint is used to mark resources for destruction and recreation. Terraform may not know that a resource is no longer valid or incorrectly configured. Taint is a way to let Terraform know. 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.
==========

- Terraform fmt applies Terraform canonical format to any tf or tfvars files. Having consistent formatting of code helps when sharing or collaborating with others. One of the terraform fmt options is -recursive. If you would like to update a collection of configurations, you can simply run the fmt command on the parent directory and it will process the contents of all subdirectories!

==========

- 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.

==========

- When you create a new repository on GitHub and create a .gitignore file, you can select Terraform as one of the options. It will automatically exclude the .terraform directory, as well as files like terraform.tfvars and terraform.tfstate. We recommend using .gitignore on any new project with those exemptions.

==========

- 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.

==========

- What is Sensitive argument?

Argument "sensitive" is an optional argument defined under output block.

```
output "db_password" {
  value       = aws_db_instance.db.password
  description = "The password for logging in to the database."
  sensitive   = true
}
```

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.

==========

```
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]
}
```

What is the index value of "1" = us-east-2

==========

- Type and Values:

The Terraform language uses the following types for its values:

string, number, bool, list or tuple, map or object.

Strings, numbers, and bools are sometimes called primitive types. Lists/tuples and maps/objects are sometimes called complex types, structural types, or collection types.

A structural type allows multiple values of several distinct types to be grouped together as a single value.

The two kinds of structural type in the Terraform language are:

* object(...): A collection of named attributes that each have their own type.
* tuple(...): A sequence of elements identified by consecutive whole numbers starting with zero, where each element has its own type.

E.g. for object and Tuple type respectively:

An object type of object({ name=string, age=number }) would match a value like the following:

```
{
 name = "h1-u"
 age  = 18
}
```

A tuple type of tuple([string, number, bool]) would match a value like the following:

["a", 15, true]

Point to Ponder: String, number, and boolean are the primitive types. Map, list, and set are the collection types. Object and tuple are the structural types. Collection types must have values of the same type, structural types can have values of different types.

REF: https://www.terraform.io/docs/configuration/expressions.html

==========

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.

==========

The lifecycle configuration block allows you to set three different flags which control the lifecycle of your resource.

- create_before_destroy - This flag is used to ensure the replacement of a resource is created before the original instance is destroyed.
- prevent_destroy - This flag provides extra protection against the destruction of a given resource.
- ignore_changes - Customizes how diffs are evaluated for resources, allowing individual attributes to be ignored through changes.

==========

In order to work with element function the variable type MUST be a "list" type. The element function 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

```
# terraform console
> element(["a", "b", "c"], 1)
b

> element(["a", "b", "c"], 3)
a
```

==========

Prevent Terraform downloading the additional plugins automatically.

```
# terraform init -input=false -get-plugins=false -plugin-dir=/usr/lib/custom-terraform-plugins
```

Where,

-plugin-dir --> Denotes plugins in the given directory are available for use
-input=false --> Do not prompt for input.
-get-plugins=false --> To prevent Terraform from automatically downloading additional plugins.

==========

Note that unlike count, splat expressions are not directly applicable to resources managed with for_each, as splat expressions are for lists only. You may apply a splat expression to values in a map like so:

values(aws_instance.example)[*].id

==========

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.

```
provider "aws" {
  alias  = "usw1"
  region = "us-west-1"
}

provider "aws" {
  alias  = "usw2"
  region = "us-west-2"
}

module "ipsec-tunnel" {
  source    = "./vpn_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"
}
```

==========

Terraform uses environment variable at number of occasions where there is a need to change Terraform default behavior .

To Enable/Disable logging:

```
export TF_LOG=TRACE                 >>>>> Enables detailed logs to appear on stderr which is useful for debugging.
export TF_LOG=                      >>>>> Disable logging
export TF_LOG_PATH=./terraform.log  >>>>> This specifies where the log should persist its output to.
```

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

```
export TF_INPUT=0  >>>>> Disable prompts for variables that haven't had their values specified.  If set to "false" or "0", causes terraform commands to behave as if the -input=false flag was specified.
```

TF_VAR_name

Environment variables can be used to set variables.

e.g.

```
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" }'
```

Must be always TF_VAR_*

TF_CLI_ARGS and TF_CLI_ARGS_name

This allows easier automation in CI environments as well as modifying default behavior of Terraform on your own system.

The flag TF_CLI_ARGS affects all Terraform commands and TF_CLI_ARGS_name will affect that specific command which you will set as an environment variable.

e.g.

```
TF_CLI_ARGS_plan="-refresh=false"
# export TF_CLI_ARGS_plan="-no-color"
```

TF_DATA_DIR

Terraform keeps all its pre-working data (such as remote backend configuration, providers, modules etc.) to ".terraform" directory by default. But using TF_DATA_DIR you can change this default behavior.