Objective 4: Use the Terraform CLI (outside of core workflow)

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

```
terraform fmt
```

- This command is used for rewriting Terraform configuration files to a canonical format and style.
- It applies the Terraform language style conventions along with other changes for readability.
- This insures consistency
- There might be changes with Terraform versions so it is recommended to run this command on modules after an upgrade.

▼ Given a scenario: choose when to use terraform taint to taint Terraform resources

```
terraform taint
```

- Marks a resource as tainted, forcing it to be destroyed and recreated on the next apply.
- It does not modify infrastructure but does modify the state file
- After a resource is marked the next plan shows it will be destroyed and recreated on the next apply
- Useful when we want a side effect of a recreation that is not visible in the attributes of the resource. For ex/rebooting the machine from a base image causing a new startup script to run.
- This command can affect resources that depend on the tainted resource. Ex/ DNS resource that uses IP of a server, that resource might need to be updated with the new IP of a tainted server.
- Examples:

  ```
  #Tainting a Single Resource
  terraform taint aws_security_group.allow_all


  #Tainting a single resource created with for_each
  terraform taint 'module.route_tables.azurerm_route_table.rt[\"DefaultSubnet\"]'
  ```

```
#Tainting a Resource within a Module
terraform taint "module.couchbase.aws_instance.cb_node[9]"
```

▼ Given a scenario: choose when to use terraform import to import existing infrastructure into our Terraform state

```
terraform import
```

- Imports existing resources into Terraform
- Examples:

  ```
  #Import into Resource
  #import an AWS instance into the aws_instance resource named foo
  terraform import aws_instance.foo i-abcd1234


  #Import into Module
  #import an AWS instance into the asw_instance resource named bar into module na
  terraform import module.foo.aws_instance.bar i-abcd1234


  #Import into Resource configured with count
  #import an AWS instance into the first instance of the aws_instance resource na
  terraform import 'aws_instance.baz[0]' i-abcd1234


  #Import into Resource configured with for_each
  #import an AWS instance into the example instance of the aws_instance resource
  terraform import 'aws_instance.baz["example"]' i-abcd1234   #Linux, MacOs, Unix
  terraform import 'aws_instance.baz[\"example\"]' i-abcd1234 #PowerShell
  terraform import aws_instance.baz[\"example\"] i-abcd1234   #Windows
  ```

▼ Given a scenario: choose when to use terraform workspace to create workspaces

```
terraform workspace
terraform workspace select
terraform workspace new
```

- Terraform configuration has a backend that defines operations and where persistent data is stored (state)

- Persistent data in the backend belongs to a workspace.
- Creating different workspaces is useful to manage different stages of deployment (sandbox or production)
- At first the backend only has one workspace 'default'. This workspace cannot be deleted.
- Certain backends can support multiple named workspaces. This allows multiple states to be associated with a single configuration.
- Config still only has one backend with more than one instance of that config
- Backends that support multiple workspaces:
  - AzureRM
  - Consul
  - COS
  - GCS
  - Local
  - Manta
  - Postgres
  - Remote
  - S3
- Examples:

  ```
   #Creating a workspace
  terraform workspace new bar
  #Created and switched to workspace "bar"!

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

▼ Given a scenario: choose when to use terraform state to view Terraform state

```
terraform state
```

- Used for advanced state management
- Used instead of changing state directly
- this is a nested subcommand (has more subcommands)
  - Resource Addressing
  - list
  - mv

  - pull
  - push
  - rm
  - show

▼ Given a scenario: choose when to enable verbose logging and what the outcome/value is

```
TF_LOG
#LOG LEVELS
TRACE
DEBUG
INFO
WARN
ERROR
TF_LOG_PATH #Persist logged output
```

- Trace is the most verbose and it is the default
- If Terraform crashes a Crash log is saved with the debug logs with panic message and backtrace

⏮Objective 3 || Objective 5 ⏩

🔙README