Hashicorp Terraform Associate Master Cheat Sheet

Objectives

- 1. Understand Infrastructure as Code (IaC) concepts
- 2. Understand Terraform's purpose (vs other IaC)
- 3. Understand Terraform basics
- 4. Use the Terraform CLI (outside of core workflow)
- 5. Interact with Terraform modules
- 6. Navigate Terraform workflow
- 7. Implement and maintain state
- 8. Read, generate, and modify configuration
- 9. Understand Terraform Cloud and Enterprise capabilities

Additional Hashicorp Resources

- The exam will be on Terraform 0.12.0 and higher.
- Exam Review
- Official Study Guide
- Lab Tutorials
- Terraform Registry find modules for deploying common infra

Additional External Resources

- General Terraform Best Practices
- Terraform Cloud Interactive Learning

NOTE: Most of this information comes directly from the Hashicorp website I've compiled the resources here for accessibility and easy review. 🙂

Objective 1: Understand Infrastructure as Code (IaC) concepts

Explain What IaC is?

Infrastructure is described using a high-level configuration syntax. This allows a blueprint of our data center to be versioned and treated as we would any other code. Additionally, infrastructure can be shared and re-used.

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

It is infrastructure (CPUs, memory, disk, firewalls, etc.) defined as code within definition files.

Describe advantages of IaC patterns?

- Can be applied throughout the infrastructure lifecycle
 - o Day 0 : Initial Build
 - Day 1 : OS and application config you apply after the initial build. Includes OS updates, patches, app config.
- Saves time by making it easy to provision and apply infrastructure configuration. Workflow is standardized across providers wether its VMWare, AWS, Azure, or GCP.
- It's easy to understand the intent of infrastructure changes.
- lac makes changes idempotent:
 - The result will always be the same since the same code is being applied
- lac makes changes consistent:
 - The manual work is removed with lac no more need for system administrators to remotely connect to each machine by executing a series of commands or scripts which can cause inconsistencies based on who executes it
- lac makes changes predictable:
 - code can be tested before applying it to production so results are always predictable
- lac allows for mutation in previously defined configurations, making for a more manageable system

Objective 2: Understand Terraform's purpose (vs other IaC)

Explain multi-cloud and provider-agnostic benefits

Multi-cloud deployment increases fault tolerance. This means in the event of failure there is a more graceful recovery of a region or provider.

The benefits of being provider-agnostic means there can be a single configuration that manages many providers.

Explain the benefits of state

- Mapping to the Real World
 - Terraform requires a database to map Tf(Terraform) config to the real world. ex.
 With state mapping Tf knows resource resource "aws_instance"
 "foo" represents instance i-abcd34233.

• Metadata

- o Tf tracks metadata or resource dependencies
- Tf keeps a copy of the most recent set of dependencies in state. So that correct order of operations can be executed even if an item is deleted from the configuration.

• Performance

- besides basic mapping Tf also keeps a cache of attribute values for all resources in the state.
- o most optional feature of state, only used to improve performance.
- small infra: for plan and apply Tf syncs all resources in state
- large infra: cache state is used because of API rate limits and querying all resources is too slow. Large infra also make use of -refresh=false and target flags

• Syncing

- o default syncing Tf stores state in a file in the current working directory
- for teams remote state is used, remote locking is utilized to avoid multiple people running Tf at the same time.

IaC with Terraform

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, <u>etc</u>) and automates the creation of those resources at the time of apply.

• Workflows

- o Scope: Establish resources that need to be created for the project
- Author: Create the configuration based on the scoped parameters with HCL
- o Initialize: run terraform init to download the provider plug-ins for the project
- Plan & Apply: run terraform plan to verify creation then terraform apply to create the resources and state files

• Advantages of Terraform

- Platform Agnostic: allows for management of a mixed environment with the same workflow
- State Management: State files are created when a project is initialized. state is used to create plans and update our infrastructure. State determines how configuration changes are measured. When a change is made, those changes are compared with the state file to determine resource creation or changes
- Operator Confidence: terraform apply allows for review before changes are applied.

Objective 3: Understand Terraform basics

Handle Terraform and provider installation and versioning

- HashiCorp Terraform Tutorial
- This tutorial goes through the process of installing Terraform and provider installation and versioning

Providers

- The primary construct of the Terraform language are resources, the behaviors of resources rely on the resource types, resource types are defined by providers.
- Providers have a set of resource types that defines which arguments are accepted, what attributes it exports, and how changes are applied to APIs.
- Providers require their own configuration for regions, authentication etc.
- Configuration
 - providers are configured with a provider block:

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

#The google provider is assumed to be the provider for the resource
type named google_compute_instance
```

- o configuration arguments like project and region are evaluated in order
- 2 meta-arguments available for provider blocks:
 - version to specify a version and
 - alias to use same provider with different config for different resources
- provider blocks are not required if not explicitly configured Tf uses an empty default config when a resource from the provider is added

Initialization

- when a new provider is added to configuration Tf has to initialize the provider before it can be used
- o terraform init downloads and initializes any providers
- only installs to current working directory, other directories can have other versions installed
- Versions

- o versions should be configured in production to avoid breaking changes
- the required_providers block should be used in the Tf block:

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

- When terraform init is re-run with providers already installed, it will use an already-installed provider that meets the constraints in preference to downloading a new version
- to upgrade all modules run terraform init -upgrade

Multiple Provider Instances

 we can have multiple configs for the same provider by using the alias metaargument to allow for multiple regions per provider, targeting multiple Docker hosts, etc.

```
0
    # The default provider configuration
       provider "aws" {
0
       region = "us-east-1"
0
       }
0
0
    # Additional provider configuration for west coast region
0
      provider "aws" {
    alias = "west"
0
0
           region = "us-west-2"
0
       }
```

• Third Party Plugins

- \circ $\,$ anyone can develop and distribute 3rd party Tf provers
- need to be manually downloaded because they are not supported by terraform init
- download must go in the user plugin directory Windows:
 %APPDATA%\terraform.d\plugins | Others: ~/.terraform.d/plugins

• Plugin Cache

- terraform init downloads plugins into a subdirectory of the working directory so each working dir is self contained. This means with more than one configuration with the same provider has a separate copy of the plugin for each config
- plugins can be large so this isn't performant Tf allows for a shared local directory for plugin cache. This has to be manually created in the CLI Configuration File.
- # (Note that the CLI configuration file is _not_ the same as the .tf files
- # used to configure infrastructure.)
 - plugin_cache_dir = "\$HOME/.terraform.d/plugin-cache"

Terraform Settings

0

• Terraform Block Syntax

- o only constant values can be used
- o terraform {
- o **# ...**
 - }

• Configuring a Terraform Backend

• this determines how state is stored, how operations are performed, remote backends for teams etc.

```
o terraform {
```

}

- backend "s3" {
- o # (backend-specific settings...)
- 0 }

0

- Specifying a Required Terraform Version
- Specifying Required Provider Versions
- Experimental Language Features

Describe plug-in based architecture

• Terraform is build on plug-in based architecture. Providers and provisioners used in configuration are plugins (AWS, Heroku). Anyone can create a new plugin.

Demonstrate using multiple providers

• Build Infrastructure- Providers

Describe how Terraform finds and fetches providers

- Resource types are defined by providers
- Provider configuration is created with a provider block, the provider name is the name in the block header
- When a new provider is added Terraform has to initialize it before its used with the terraform init command. This downloads and installs the providers plugin

Explain when to use and not use provisioners and when to use local-exec or remote-exec

- Provisioners provisioners are used to model specific actions on the local machine or on a remote machine to prepare infrastructure objects
- Provisioners are there if needed but they add complexity and uncertainty (should only be used as a last result)
- Provisioners should be used if no other option will work.
- Use cases:
 - o Passing data into virtual machines and other compute resources
 - running config management software
- local-exec invokes a local executable after the resource is created. Invokes a process on the machine not on the resource.

remote-exec - invokes a script on a remote resource after it is created.

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

Given a scenario: choose when to use terraform fmt to format code terraform $\ensuremath{\mathsf{fmt}}$

- 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 named foo: 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 named baz configured with count: 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 named baz configured with for_each: 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
 - o Consul
 - COS
 - \circ GCS
 - Local
 - o Manta
 - Postgres
 - o 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
 - o list
 - o mv
 - o pull
 - o push
 - o rm
 - o 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 5: Interact with Terraform modules

Contrast module source options

- Module Overview
 - Definition a set of configuration files in a single directory. A container for multiple resources that are used together.
 - A module that is called by another configuration is sometimes referred to as a "child module" of that configuration.
- Applications
 - Organize configuration easier to navigate, understand, and update our configuration by keeping all related parts together.
 - Encapsulate configuration put configuration into distinct logical components. Reduces chance of error. Ex/naming two diff resources the same thing.

- Re-use configuration share and re-use modules with the public and teams
- Provide consistency and ensure best practices
- Module source options:
 - we reference a **Public Registry Module** by declaring the source.

```
    module "consul" {
```

```
• #<NAMESPACE>/<NAME>/<PROVIDER>
```

```
    source = "hashicorp/consul/aws"
```

```
• version = "0.1.0"
}
```

o Private Registry Module Sources follow this syntax

```
module "vpc" {
  #<HOSTNAME>/<NAMESPACE>/<NAME>/ <PROVIDER>
  source = "app.terraform.io/example_corp/vpc/aws"
  version = "0.9.3"
}
```

Interact with module inputs and outputs Modules

Describe variable scope within modules/child modules

- variables are parameters for modules
- variables allow us to customize modules without changing the source code and they allow for modules to be shared between different configurations.
- root module variables can be set with CLI and environment variables.
- When declaring variables in child modules, the calling module should pass values in the module block.
- Declaring a variable:
- variable names have to be unique per module
- any name can be used except for :source, version, providers, count,for_each,lifecycle,depends_on,locals
- Note: if type and default are used, default must be convertible to the type
- variable "image_id" {
- type = string
- #defines what value types are accepted for the variable, if not explicit any type is accepted.
- #Types: string,number,bool, any(to allow for any type) | Complex Type: list(<TYPE>),set(<TYPE>),map(<TYPE>),object({<ATTR NAME> = <TYPE>, ... }), tuple([<TYPE>, ...])

```
    validation {
```

- condition = length(var.image_id) > 4 && substr(var.image_id, 0, 4) == "ami-"
- error_message = "The image_id value must be a valid AMI id, starting with \"ami-\"."

```
• }
```

• #validation rules are experimental - uses value of variable to return true or false

• }

```
variable "availability_zone_names" {
        = list(string)
  type
•
   default = ["us-west-1a"]
   #default means the variable is considered optional, used if no other value
•
   is set when calling the module or running Terraform
   description = "variable description, purpose and value expected"
•
    }
   variable "docker_ports" {
•
   type = list(object({
     internal = number
     external = number
     protocol = string
     }))
   default = [
     {
       internal = 8300
       external = 8300
       protocol = "tcp"
    }
   ]
•
   }
•
•
   #-----
  #To use validation we need to opt in
  terraform {
   experiments = [variable_validation]
•
   }
   Using variable values
     resource "aws_instance" "example" {
     instance type = "t2.micro"
     ami = var.image_id #expression reads var.<NAME> name is the label
   declared on the variable
     }
```

- Set root module variables 1) In Terraform Cloud Workspace 2) Individual CLI with var 3) In .tfvars file 4) As environment variable
- child modules have variables set in the configuration of the parent module

Discover modules from the public Terraform Module Registry

- Finding and Using Modules
 - Terraform Registry

Defining module version

- Use the version attribute in the module block to specify versions:
- module "consul" {
- source = "hashicorp/consul/aws"
- version = "0.0.5" #single explicit version
- #or
- version = >= 1.2.0 #version constraint expression

```
• servers = 3
}
```

Objective 6: Navigate Terraform workflow

Describe Terraform workflow (Write -> Plan -> Create)

- Write
 - o Author infrastructure as code
- Plan
 - Preview changes before applying
- Create (Apply)
 - Provision reproducible infrastructure
- Configuration is written like any program, use version control to keep track of changes
- # Create repository
- \$ git init my-infra && cd my-infra
- Initialized empty Git repository in /.../my-infra/.git/
- # Write initial config
- \$ vim main.tf
- # Initialize Terraform
- \$ terraform init
- Initializing provider plugins...
- # ...
 - Terraform has been successfully initialized!
- running Terraform plan repeatedly is useful to make sure there are no syntax errors and the correct code is being written per the desired outcome.
- First run Terraform apply before pushing to git to make sure the provisions are correct
- While working in teams it is best to use branches to avoid code collision.
- \$ git checkout -b <branch-name>
 - Switched to a new branch <branch-name>
- Teams can review changes via Terraform plans and pull requests
- Terraform cloud helps streamline this process in a team setting
 - Write secure location for storing variables and state with the "remote" backend, then a Terraform Cloud API key is used to edit the configuration and run plans against the state file.

```
    terraform {
```

```
    backend "remote" {
```

```
organization = "my-org"
```

```
workspaces {
```

```
prefix = "my-app-"
```

- }
- }
- } • #-----
- #-----
- \$ terraform workspace select my-app-dev
- Switched to workspace "my-app-dev".
- \$ terraform plan
- Running plan remotely in Terraform Enterprise.
- Output will stream here. To view this plan in a browser, visit:
- https://app.terraform.io/my-org/my-app-dev/.../

- Refreshing Terraform state in-memory prior to plan...
- # ...
 Plan: 1 to add, 0 to change, 0 to destroy.
 - Plan plans are automatically run when a pull request is created. Status updates are shown in the pull request view.
 - Apply A confirm and apply is needed after merging to run an apply.

The next section will go over Terraform Commands

Initialize a Terraform working directory (terraform init)

terraform init

- prepares working directory for use
- safe to run multiple times to bring the working directory up to date
- it will never delete a configuration or state

Validate a Terraform configuration (terraform validate)

terraform validate

- validates the configuration files in the dir, this does not apply to things like remote state or provider APIs
- validate checks for syntax, internal consistency, such as attribute names and value types
- safe to run automatically or as a test step for Cl
- requires initialized working directory

Generate and review an execution plan for Terraform (terraform plan)

Terraform plan

• Creates an execution plan, automatically performs a refresh

Execute changes to infrastructure with Terraform (terraform apply)

terraform apply

- applies changes needed for the desired state of the configuration
- runs set of actions defined by a terraform plan command

Destroy Terraform managed infrastructure (terraform destroy)

terraform destroy

completely destroys and terraform created infrastructure

Objective 7: Implement and maintain state

Describe default local backend

- Backends by default Terraform uses 'local' backend
 - This is an abstraction that determines how state is loaded and how an operation is executed. It allows such actions as non-local file state storage and remote execution
 - Benefits:
 - Working in a team can store state remotely and use locks to prevent corruption in state
 - Keeping sensitive information off disk state in backends are only stored in memory
 - Remote operations terraform apply can take time for larger infrastructures, some backends can use remote operations instead to execute commands remotely
- Local example config:
- terraform {

}

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

```
•
}
```

Outline state locking State Locking

- if supported by your backend state can be locked so others cannot change it while another change is being made.
- this is automatic for all operations that can write state
- Backends types supporting locking:(standard)artifactory,azurerm,consul,cos,etcd,etcdv3,gcs,http,manta,oss,pg,s3,swift ,terraform enterprise, and in enhanced backends there are remote operations as well (plan, apply, etc.)
- A lock can be forced open with force-unlock which requires a unique nonce lock ID

Handle backend authentication methods

• Different backends have different configuration for authentication, authentication can be done different ways within a backend.

```
• Example with azurerm:
```

```
• #authenticating using the Azure CLI or a Service Principal:
```

```
    terraform {
    backend "azurerm" {
    resource_group_name = "StorageAccount-ResourceGroup"
```

key

}

```
    storage_account_name = "abcd1234"
```

```
container_name = "tfstate"
```

```
= "prod.terraform.tfstate"
```

```
• }
```

```
• #-----
```

#authenticating using Managed Service Identity (MSI):

•

```
terraform {
•
        backend "azurerm" {
            storage_account_name = "abcd1234"
            container_name = "tfstate"
            key
                              = "prod.terraform.tfstate"
•
                              = true
•
            use msi
            subscription_id = "0000000-0000-0000-0000-0000000000"
•
            tenant_id = "0000000-0000-0000-0000-00000000000"
            }
    }
```

Describe remote state storage mechanisms and supported standard backends **Remote State Storage**

- Uses Terraform Cloud as a backend, allows free remote state management
- Tutorial for Remote State Storage

Standard backends

 artifactory,azurerm,consul,cos,etcd,etcdv3,gcs,http,manta,oss,pg,s3,swift,terraform enterprise

Describe effect of Terraform refresh on state

- terraform refresh
- reconciles the state Terraform knows about via the state file.
- refresh does not modify the infrastructure, it does modify the state file.

Describe backend block in configuration and best practices for partial configurations Backend Config

- Backends are configured in the Terraform files.
- there can only be one backend
- This is an example of a config for "consul":

```
terraform {
   backend "consul" {
    address = "demo.consul.io"
    scheme = "https"
   path = "example_app/terraform_state"
  }
}
```

Partial Configuration

- You can omit certain arguments from the backend configuration.
- This is done to avoid storing access keys or private data in the main configuration
- adding the omitted arguments must be done during the initialization process by doing the following:
 - o Interactively If interact input is enabled it will ask you for the required values
 - File terraform init -backend-config=PATH that contains the variables

 Command-link key/value pairs - terraform init -backendconfig="KEY=VALUE" **This isn't recommended for secret keys since CL flags can be stored in a history file.

Understand secret management in state files

- state contains resource IDs and attributes, db data that may have passwords.
- with remote state, state is only in memory when in use. This is more secure
- also some backends can encrypt the state data at rest
- Terraform Cloud encrypts state at rest and protects it with TLS in transit.
- Terraform Cloud keeps track of user identity, and state changes.

Objective 8: Read, generate, and modify configuration

Demonstrate use of variables and outputs Input Variables Tutorial

Output Variables Tutorial

Describe secure secret injection best practice **Vault Provider for Terraform**

- Best Practices
 - o avoid putting secret or sensitive variables in config or state files.
 - Webinar walk-through on Best Practices
 - o set secret variables for provider config block in environment variables.

```
#auth_login Usage with userpass backend
variable login_username {}
variable login_password {}
provider "vault" {
  auth login {
    path = "auth/userpass/login/${var.login_username}"
    parameters = {
      password = var.login_password
    }
  }
}
#auth_login Usage with approle
variable login_approle_role_id {}
variable login_approle_secret_id {}
provider "vault" {
  auth_login {
    path = "auth/approle/login"
    parameters = {
      role_id = var.login_approle_role_id
```

```
secret_id = var.login_approle_secret_id
    }
  }
}
#For multiple namespace in vault use alias
provider "vault" {
  alias = "ns1"
  namespace = "ns1"
}
provider "vault" {
   alias = "ns2"
  namespace = "ns2"
}
resource "vault_generic_secret" "secret"{
  provider = "vault.ns1"
  . . .
}
```

Understand the use of collection and structural types Complex Types

Collection Types

complex types group values into a single value. 2 types: Collection type(grouping similar values) and Structure types (grouping dissimilar values)

Structural Types

	20000000 - J F
multiple values of a type can be grouped together. The type of value within a collection is called element type	multiple values of several types grouped together
Example:list(string)List of string	Example: Object type of object({ name=string, age=number }) would match this value: { name "John" age = 52 } Example of tuple:["a", 15, true]
Collection Types:list():Sequence of whole numbers starting at 0 map():collection of values id'd by a label set():unique values with no ids or order	<pre>Structural Types: object():collection of named attributes that have their own type.The schema for object types is {</pre>

Create and differentiate resource and data configuration

•	Syntax	Types and Arguments	Behavior	Meta- Arguments
Resources	blocks declare a resource of a given type aws_instance with a local name web. The local name is used to reference the resource in the module. In the braces {} config arguments are defined for the resource type.	each resource has a single resource type, each type belongs to a provider, body of resource are specific to type	when you create a new resource it only exists in the configuration until you apply it. When its created it is saved in state, and can be updated or destroyed	Each resource is associated with a single resource type, which determines the kind of infrastructure object it manages and what arguments and other attributes the resource supports
Data Sources	A data source is accessed via a special kind of resource known as a data resource, declared using a data block	Each data resource is associated with a single data source, this determines the kind of object(s) it reads and the available arguments. Most of the items within the body of a data block are defined by and specific to the selected data source, and these arguments can make	If the query constraint arguments for a data resource refer only to constant values or values that are already known, the data resource will be read and its state updated during Terraform's "refresh" phase, which runs prior to creating a plan. more on behavior	As data sources are essentially a read only subset of resources, they also support the same meta- arguments of resources with the exception of the lifecycle configuration block.

Syntax	Types and Arguments	Behavior	Meta- Arguments
	full use of expressions and other dynamic Terraform language features.		

Use resource addressing and resource parameters to connect resources together Connecting resources

Use Terraform built-in functions to write configuration Built-in Functions

- Terraform only supports given functions
- List of Functions
- This can also be viewed in the repo here
- To test functions in the command line run terraform console

Configure resource using a dynamic block Dynamic Blocks

- In top level block constructs(like resources) expressions can be used only when assigning a value to an argument with name=expression
- Some resource types have repeatable nested blocks in their arguments that don't accept expressions.

```
• Example:
```

```
    resource "aws_elastic_beanstalk_environment" "tfenvtest" {
    name = "tf-test-name" # can use expressions here
    setting {
```

```
# but the "setting" block is always a literal block
}
```

```
• }
```

• You can create repeatable nested blocks with the block type dynamic. This is supported with resource, data, provider, and provisioner blocks

```
• Example:
```

```
    resource "aws_elastic_beanstalk_environment" "tfenvtest" {
```

```
name = "tf-test-name"
```

```
application = "${aws_elastic_beanstalk_application.tftest.name}"
```

```
    solution_stack_name = "64bit Amazon Linux 2018.03 v2.11.4 running Go
1.12.6"
```

```
    dynamic "setting" {
```

```
    for_each = var.settings
```

```
• content {
```

```
    namespace = setting.value["namespace"]
```

```
name = setting.value["name"]
```

```
value = setting.value["value"]
}
```

- }
- Dynamic blocks can only produce arguments that belong to the resource type, data source, provider or provisioner being configured.
- Overuse of dynamic blocks can get hard to read, it's recommended to use them only to hide details in order to build a clean user interface for re-usability.

Describe built-in dependency management (order of execution based) Resource Dependencies Tutorial

Objective 9: Understand Terraform Cloud and Enterprise capabilities

Describe the benefits of Sentinel, registry, and workspaces

- Sentinel
 - An embedded policy as a code framework used with Enterprise products. Policies written in the Sentinel language
 - Used for logic based policy decisions and can be extended to use information form external sources.
 - tfe_sentinel_policy resource can be used to upload a policy using Terraform itself
 - Sentinel can be used with Terraform Cloud as well by:
 - Defining the policies Policies are defined using the policy language with imports for parsing the Terraform plan, state and configuration.
 - Managing policies for organizations Users with permission to manage policies can add policies to their organization by configuring VCS integration or uploading policy sets through the API. They also define which workspaces the policy sets are checked against during runs. (More about permissions.)
 - Enforcing policy checks on runs Policies are checked when a run is performed, after the terraform plan but before it can be confirmed or the terraform apply is executed.
 - Mocking Sentinel Terraform data Terraform Cloud provides the ability to generate mock data for any run within a workspace. This data can be used with the Sentinel CLI to test policies before deployment.

Module Registry

 Private module registry helps us share Terraform modules with other organizations.

- Support includes module versioning, search and filtering list of modules, and a configuration designer to build workspaces
- Similar to the Public Registry
- o Module Registry Announcement
- Workspaces
 - o Using Workspaces is how Terraform Cloud organized infrastructure
 - Workspaces are Collections of Infrastructure usually organizations have to manage many collections.
 - Each collection contains a configuration, state data, and variables.
 - Terraform Cloud manages infrastructure collections with workspaces instead of directories. A workspace contains everything Terraform needs to manage a given collection of infrastructure, and separate workspaces function like completely separate working directories.
 - o Terraform Enterprise: Understanding Workspaces And Modules

Differentiate OSS (Open Source Software) and Terraform Cloud workspaces

- CLI Workspaces
 - relates to persistent data stored in the backend, a feature for using one configuration to manage many similar grouped resources.
 - o uses with a Terraform's command line interface: terraform workspace new bar
- Enterprise/Cloud Workspaces
 - Uses Workspaces to manage break up parts of a system

Summarize features of Terraform Cloud

- Terraform Cloud
 - o Terraform Cloud Pricing/Features
 - o Terraform Cloud Docs
 - Main Features
 - Workflow
 - CLI, Remote Execution(Operations), Support for Local Execution, Organize infra with workspaces, Remote state management, data sharing, and run triggers, version control integration, private module registry
 - Integrations
 - Full API, Notifications
 - Access Control and Governance
 - team based permission systems, sentinel policies, cost estimation