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
  - avoid putting secret or sensitive variables in config or state files.
  - Webinar walk-through on Best Practices
  - 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

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

Collection Types	Structural Types
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	Structural Types: `object()` :collection of named attributes that have their own type.The schema for object types is `{ <KEY> = <TYPE>, <KEY> = <TYPE>, ... }` and `tuple()` :sequence of elements id'd by whole numbers, each element has its own type.The schema for tuple types is `[<TYPE>, <TYPE>, ...]`

▼ Create and differentiate resource and data configuration

- Code Examples for Resources and Data Sources

	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	Resource behavior can be changed with the use of meta-arguments
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	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	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
		by and specific to the selected data source, and these arguments can make full use of expressions and other dynamic Terraform language features.	"refresh" phase, which runs prior to creating a plan. more on behavior	

▼ 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 7 || Objective 9 ⏩

🔙README