TERRAFORM SYNTAX

➢ TERRAFORM SYNTAX:

- The majority of the Terraform language documentation focuses on the practical uses of the language and the specific constructs it uses.
- Style Conventions documents some commonly accepted formatting guidelines for Terraform code. These conventions can be enforced automatically with **terraform fmt.**
- There are two types of syntaxes:
  - Configuration Syntax
  - JSON Configuration Syntax

CONFIGURATION SYNTAX:

- Configuration Syntax describes the native grammar of the Terraform language.
- This low-level syntax of the Terraform language is defined in terms of a syntax called HCL, which is also used by configuration languages in other applications, and in particular other HashiCorp products.
- The Terraform language syntax is built around two key syntax constructs:
  - Arguments
  - Blocks

ARGUMENTS:

- An argument assigns a value to a particular name:

```
ami             =  "ami-079db87dc4c10ac91"
instance_type   =  "t2.micro"
```

- The context where the argument appears determines what value types are valid but many arguments accept arbitrary expressions.

BLOCKS:

- A block is a container for other content:

```
resource "aws_instance" "example" {
    ami = "abc123"
```

```
network_interface {
# ...
        }
}
```

- A block has a type (resource in this example).
- Each block type defines many labels must follow the **type keyword**.
- The resource block type expects two labels, which are **aws_instance** and **example** in the example above.
- The Terraform language uses a limited number of top-level block types.
- Most of Terraform's features (including **resources, input variables, output values, data sources, etc**.) are implemented as top-level blocks.
- A particular block type may have any number of required labels, or it may require none as with the nested network_interface block type.

IDENTIFIERS:

- Argument names, block type names, and the names of most Terraform-specific constructs like resources, input variables, etc. are all identifiers.
- Identifiers can contain **letters, digits, underscores (_), and hyphens (-).**
- The first character of an identifier must not be a digit, to avoid ambiguity with literal numbers.

COMMENTS:

- The Terraform language supports three different syntaxes for comments:
  - # begins a single-line comment, ending at the end of the line.
  - // also begins a single-line comment, as an alternative to #.
  - /* and */ are start and end delimiters for a comment that might span over multiple lines.

NOTE: The # single-line comment style is the default comment style and should be used in most cases. Automatic configuration formatting tools may automatically transform // comments into # comments, since the double-slash style is not idiomatic.

JSON CONFIGURATION SYNTAX:

- Most Terraform configurations are written in the native Terraform language syntax, which is designed to be relatively easy for humans to read & update.
- JSON Configuration Syntax documents how to represent Terraform language constructs in the pure JSON variant of the Terraform language. Terraform's JSON syntax is unfriendly to humans, but can be very useful when generating infrastructure as code with other systems that don't have a readily available HCL library.
- Terraform also supports an alternative syntax that is JSON-compatible. This syntax is useful when generating portions of a configuration programmatically, since existing JSON libraries can be used to prepare the generated configuration files.
- Terraform expects native syntax for files named with a **.tf** suffix, and JSON syntax for files named with a **.tf.json** suffix.

JSON FILE STRUCTURE:

- At the root of any JSON-based Terraform configuration is a JSON object. The properties of this object correspond to the top-level block types of the Terraform language. For example:

```
{
  "variable": {
    "example": {
      "default": "hello"
    }
  }
}
```

- resource blocks expect two labels, so two levels of nesting are required:

```
{
  "resource": {
    "aws_instance": {
      "example": {
        "instance_type": "t2.micro",
        "ami": "ami-abc123"
      }
    }
  }
}
```

- Taken together, the above two configuration files are equivalent to the following blocks in the native syntax:

```
variable "example" {
  default = "hello"
}

resource "aws_instance" "example" {
  instance_type = "t2.micro"
  ami           = "ami-abc123"
}
```

- Within each top-level block type the rules for mapping to JSON are slightly different.