15 Days of Terraform – Curated by Shaik Hari Sadia Anjum

Variables

Terraform variables allow you to **parameterize** your Terraform configuration. Instead of hardcoding values like region names, instance types, or AMI IDs, you use variables to make your configurations **reusable, dynamic, and cleaner**.

Types of Variables in Terraform

1. **Input Variables** – Values you pass into Terraform.
2. **Output Variables** – Values Terraform gives back after applying.

Input Variables:

Input variables are used to parameterize your Terraform configurations. They allow you to pass values into your modules or configurations from the outside. Input variables can be defined within a module or at the root level of your configuration. Here's how you define an input variable:

```
variable "region" {
 description = "The AWS region to deploy resources"
 type        = string
 default     = "us-east-1"
}
```

*You can use different **data types**:*

- *string*
- *number*
- *bool*
- *list(<TYPE>)*
- *map(<TYPE>)*
- *object({ key = type })*
- *any*

```hcl
variable "instance_type" {
  type        = string
  description = "EC2 Instance type"
  default     = "t2.micro"
}

variable "allowed_ports" {
  type = list(number)
  default = [22, 80, 443]
}
```

You refer to variables like this:

```hcl
provider "aws" {
  region = var.region
}

resource "aws_instance" "web" {
  ami           = var.ami_id
  instance_type = var.instance_type
}
```

Output Variables:

Output variables allow you to expose values from your module or configuration, making them available for use in other parts of your Terraform setup. Here's how you define an output variable:

```hcl
output "instance_id" {
  value = aws_instance.web.id
}
```

In this example:

- output is used to declare an output variable named ***instance_id***.

- description provides a description of the output variable(optional)
- value specifies the value that you want to expose as an output variable. This value can be a resource attribute, a computed value, or any other expression.

You can reference output variables in the root module or in other modules by using the syntax ***module.module_name.output_name***, where ***module_name*** is the name of the module containing the output variable.

📂 .tfvars File:

*A .tfvars file is a **variable definition file** in Terraform. It's used to **assign values to input variables** defined in your Terraform configuration.*

*Instead of passing variables on the command line every time or hardcoding them in .tf files, you place them in a **.tfvars** file for better organization, readability, and reuse.*

1. **Separation of Configuration from Code**: Input variables in Terraform are meant to be configurable so that you can use the same code with different sets of values. Instead of hardcoding these values directly into your .tf files, you use .tfvars files to keep the configuration separate. This makes it easier to maintain and manage configurations for different environments.
2. **Sensitive Information**: .tfvars files are a common place to store sensitive information like API keys, access credentials, or secrets. These sensitive values can be kept outside the version control system, enhancing security and preventing accidental exposure of secrets in your codebase.
3. **Reusability**: By keeping configuration values in separate .tfvars files, you can reuse the same Terraform code with different sets of variables. This is useful for creating infrastructure for different projects or environments using a single set of Terraform modules.
4. **Collaboration**: When working in a team, each team member can have their own .tfvars file to set values specific to their environment or workflow. This avoids conflicts in the codebase when multiple people are working on the same Terraform project.

📂 *File Structure and Naming*

✅*Common naming patterns:*

- *terraform.tfvars → loaded **automatically** by Terraform*
- *dev.tfvars, prod.tfvars, etc. → used for **multiple environments***
- *variables.auto.tfvars → automatically loaded by Terraform*

Example:

Step 1: Define variables in variables.tf

```
variable "region" {
  type        = string
  description = "AWS region"
}

variable "instance_type" {
  type        = string
  description = "EC2 instance type"
}

variable "environment" {
  type        = string
  description = "Deployment environment"
}

variable "tags" {
  type = map(string)
  description = "Tags to apply to resources"
}
```

Step 2: Assign values in .tfvars

terraform.tfvars or dev.tfvars

```
region       = "us-east-1"
instance_type = "t3.micro"
environment   = "development"
tags = {
```

```
  Owner   = "Anjum"

  Purpose = "Testing"

}
```

Step 3: Run Terraform

Automatically loads terraform.tfvars:

1.terraform apply

2.terraform apply -var-file="dev.tfvars"

Secure Handling of Secrets in .tfvars

Don't store sensitive info like passwords, tokens, or keys in plain .tfvars files. Instead:

- *Use environment variables (TF_VAR_...)*
- *Use encrypted secrets manager (AWS SSM, HashiCorp Vault)*
- *Use sensitive = true for such variables in variables.tf*

Why Use .tfvars?

Benefit	Description
🔁 Reusability	Manage multiple environments (dev.tfvars, prod.tfvars)
🔒 Security	Keeps variables separate from source code
🧼 Clean Code	Reduces clutter in your main .tf files
💡 Simplicity	Easy to change values without touching the logic

Mistake	*Problem*	*Solution*
Using wrong type	*"22" instead of 22 for port numbers*	*Match declared type in variables.tf*
Forgetting -var-file	*File doesn't auto-load unless it's terraform.tfvars or *.auto.tfvars*	*Use -var-file="filename.tfvars"*
Conflicting variable names	*Overwrites unintended variables*	*Be consistent and organized in naming*
Including secrets	*Exposing passwords in code repo*	*Use tools like Vault, SSM, or TF_VAR_password env vars*

Conditional Expressions:

Conditional expressions in Terraform are used to define conditional logic within your configurations. They allow you to make decisions or set values based on conditions. Conditional expressions are typically used to control whether resources are created or configured based on the evaluation of a condition.

The syntax for a conditional expression in Terraform is:

condition ? true_result : false_result

- ***condition is an expression that evaluates to either true or false.***
- ***true_result is the value that is returned if the condition is true.***
- ***false_result is the value that is returned if the condition is false.***

Conditional Resource Creation Example:

```
resource "aws_instance" "example" {

  count = var.create_instance ? 1 : 0


  ami           = "ami-XXXXXXXXXXXXXXXXX"

  instance_type = "t2.micro"
```

}

In this example, the count attribute of the aws_instance resource uses a conditional expression. If the create_instance variable is true, it creates one EC2 instance. If create_instance is false, it creates zero instances, effectively skipping resource creation.

Conditional Variable Assignment Example

```
variable "environment" {
 description = "Environment type"
 type       = string
 default    = "development"
}

variable "production_subnet_cidr" {
 description = "CIDR block for production subnet"
 type       = string
 default    = "10.0.1.0/24"
}

variable "development_subnet_cidr" {
 description = "CIDR block for development subnet"
 type       = string
 default    = "10.0.2.0/24"
}

resource "aws_security_group" "example" {
 name       = "example-sg"
 description = "Example security group"

 ingress {
```

```
  from_port   = 22

  to_port     = 22

  protocol    = "tcp"

  cidr_blocks = var.environment == "production" ? [var.production_subnet_cidr] :
[var.development_subnet_cidr]

 }

}
```

In this example, the locals block uses a conditional expression to assign a value to the subnet_cidr local variable based on the value of the environment variable. If environment is set to "production", it uses the production_subnet_cidr variable; otherwise, it uses the development_subnet_cidr variable.

Conditional Resource Configuration

```
resource "aws_security_group" "example" {

 name = "example-sg"

 description = "Example security group"


 ingress {

  from_port   = 22

  to_port     = 22

  protocol    = "tcp"

  cidr_blocks = var.enable_ssh ? ["0.0.0.0/0"] : []

 }

}
```

In this example, the ingress block within the aws_security_group resource uses a conditional expression to control whether SSH access is allowed. If enable_ssh is true, it allows SSH traffic from any source ("0.0.0.0/0"); otherwise, it allows no inbound traffic.

Conditional expressions in Terraform provide a powerful way to make decisions and customize your infrastructure deployments based on various conditions and variables. They enhance the flexibility and reusability of your Terraform configurations.