Variables

Terraform

Agenda

- Variables in Terraform
- Input Variables
- Local Variables
- Outputs
- Interpolation
- Conditions
- Iterations
- Data Sources

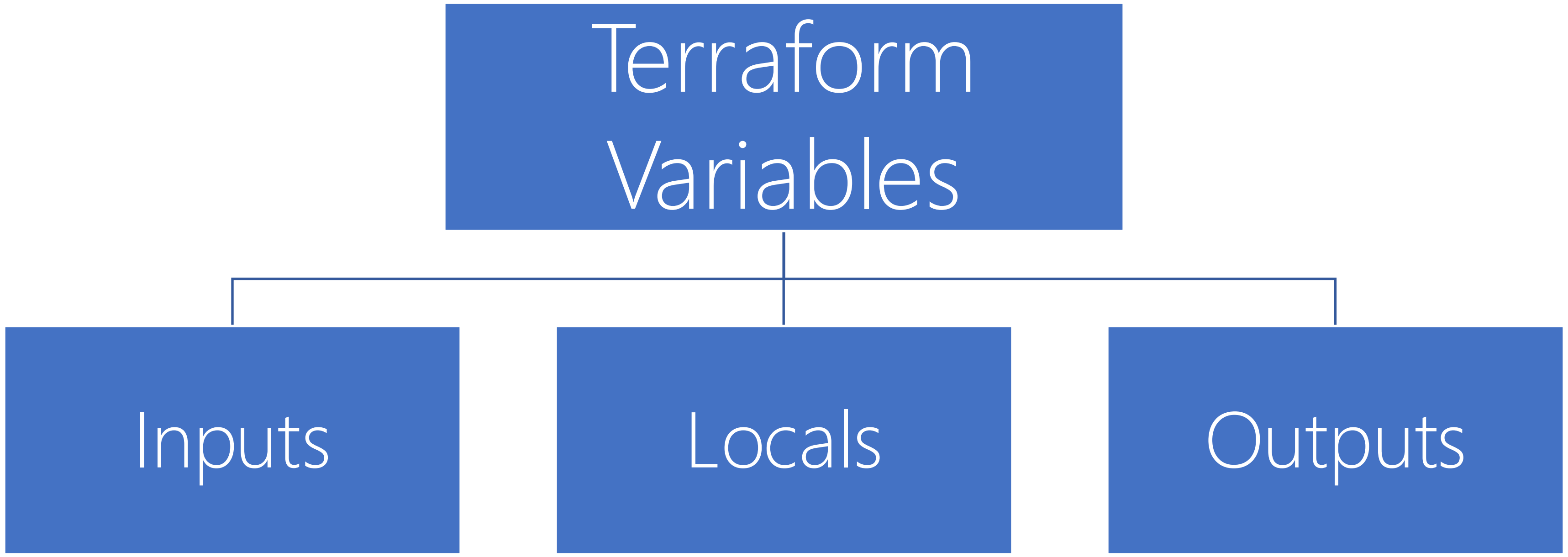
Variables in Terraform

Terraform

Variables in Terraform

- Variable a symbolic name associated with a value and whose associated value may be changed
- Variables in Terraform are a great way to define centrally controlled reusable values
- The information in Terraform variables is saved independently from the deployment plans which makes the values easy to read and edit from a single file

Variables in Terraform



Input Variables

Terraform

Input Variables

- Input variables serve as parameters for Terraform code
- This approach allows aspects of the code to be customized without altering the source code
- When you declare variables, you can set their values using CLI options, environment variables or files
- Can be accessed using the code **var.variable_name**

Input Variables Types

- String
- Number
- List
- Map
- Boolean
- Object

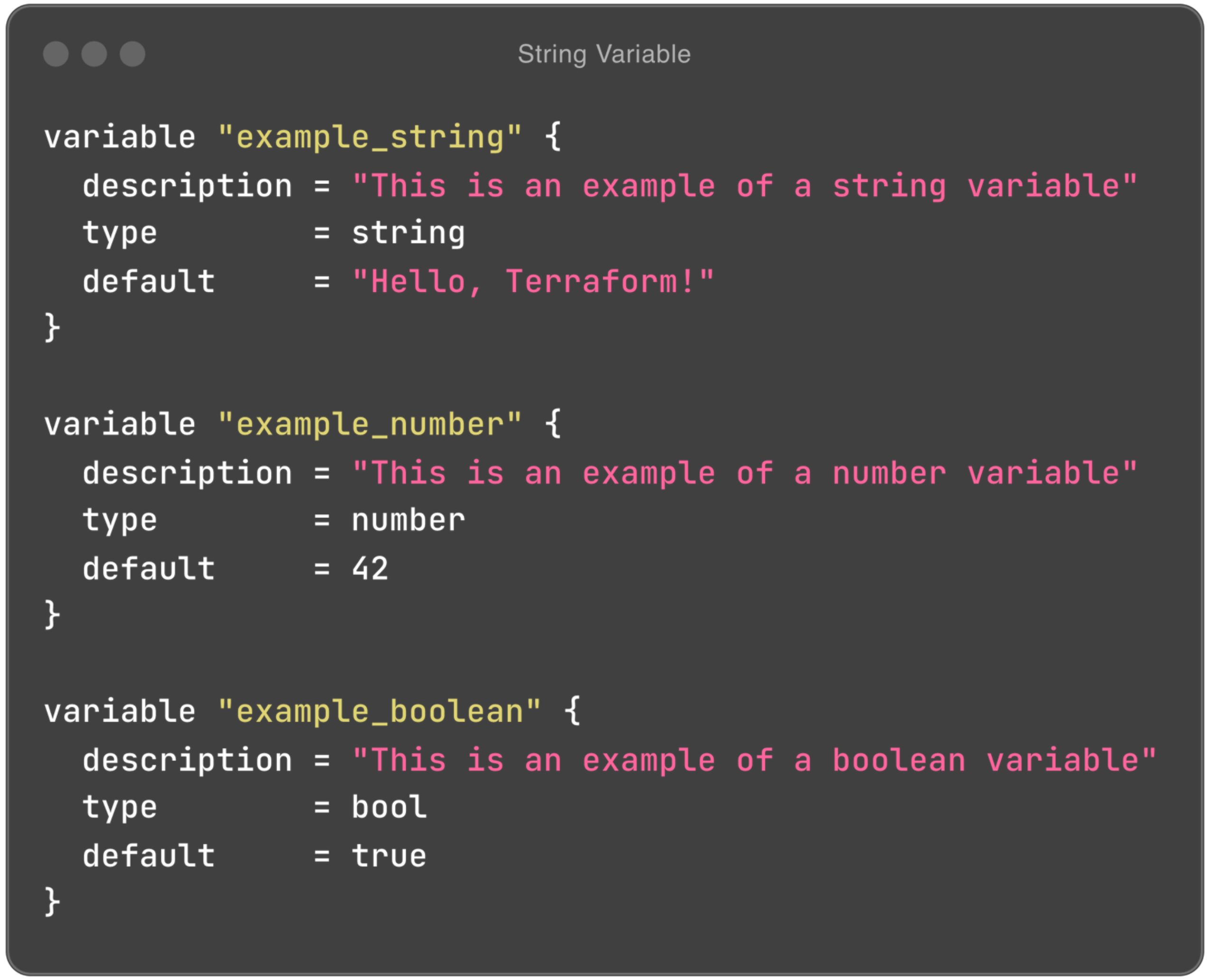
Input Variables Optional Arguments

- **default** – A default value which then makes the variable optional
- **type** – This argument specifies what value types are accepted for the variable
- **description** – This specifies the input variable documentation
- **validation** – A block to define validation rules usually in addition to type
- **sensitive** – Limits Terraform output when the variable is used

Input Variables Optional Arguments

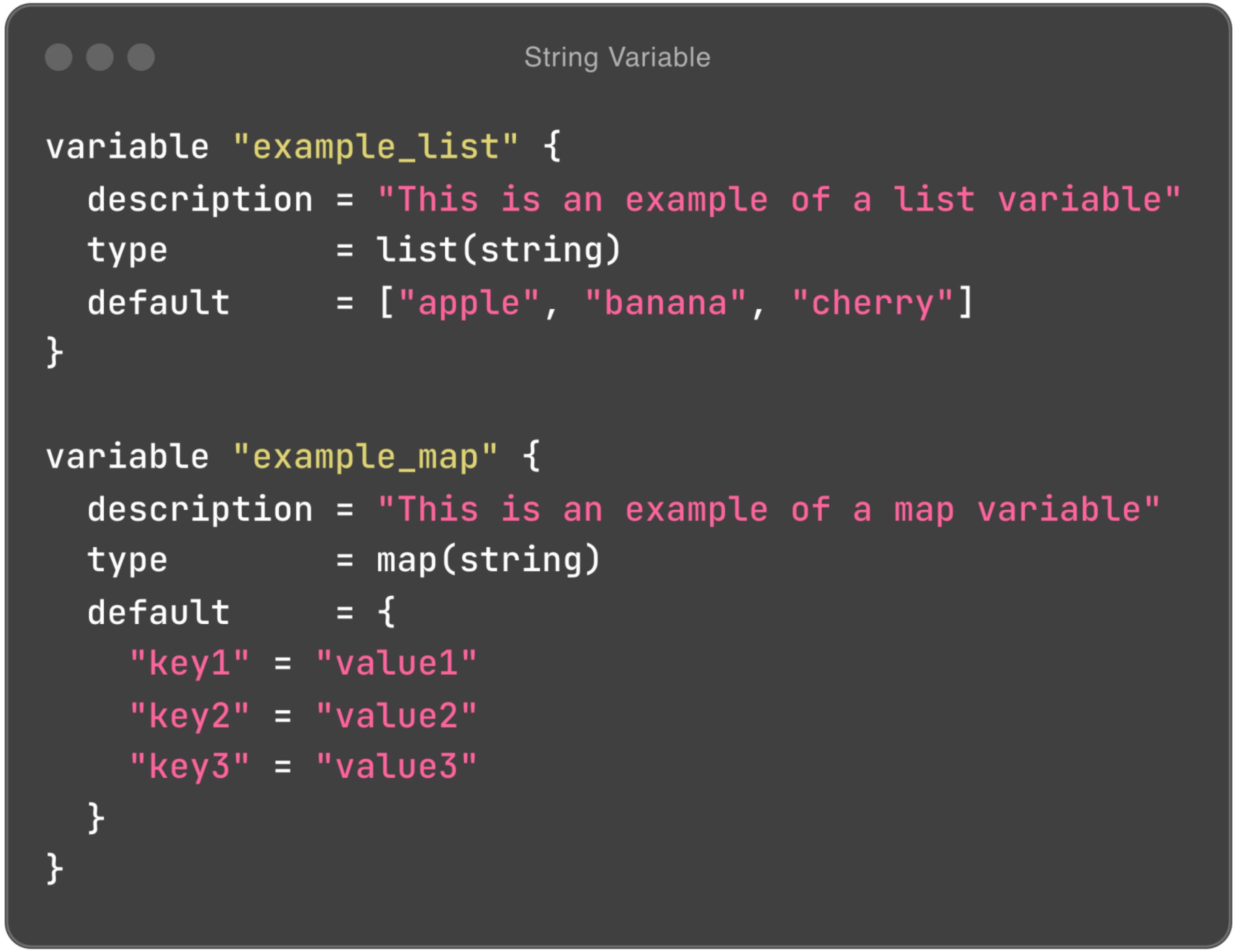
- As a recommendation setting a description and type for all variables
- Setting a default value when practical
- If you do not set a default value for a variable, you must assign a value before Terraform can apply the configuration
- Terraform does not support unassigned variables

Input Variables: String, Number, Boolean


String Variable

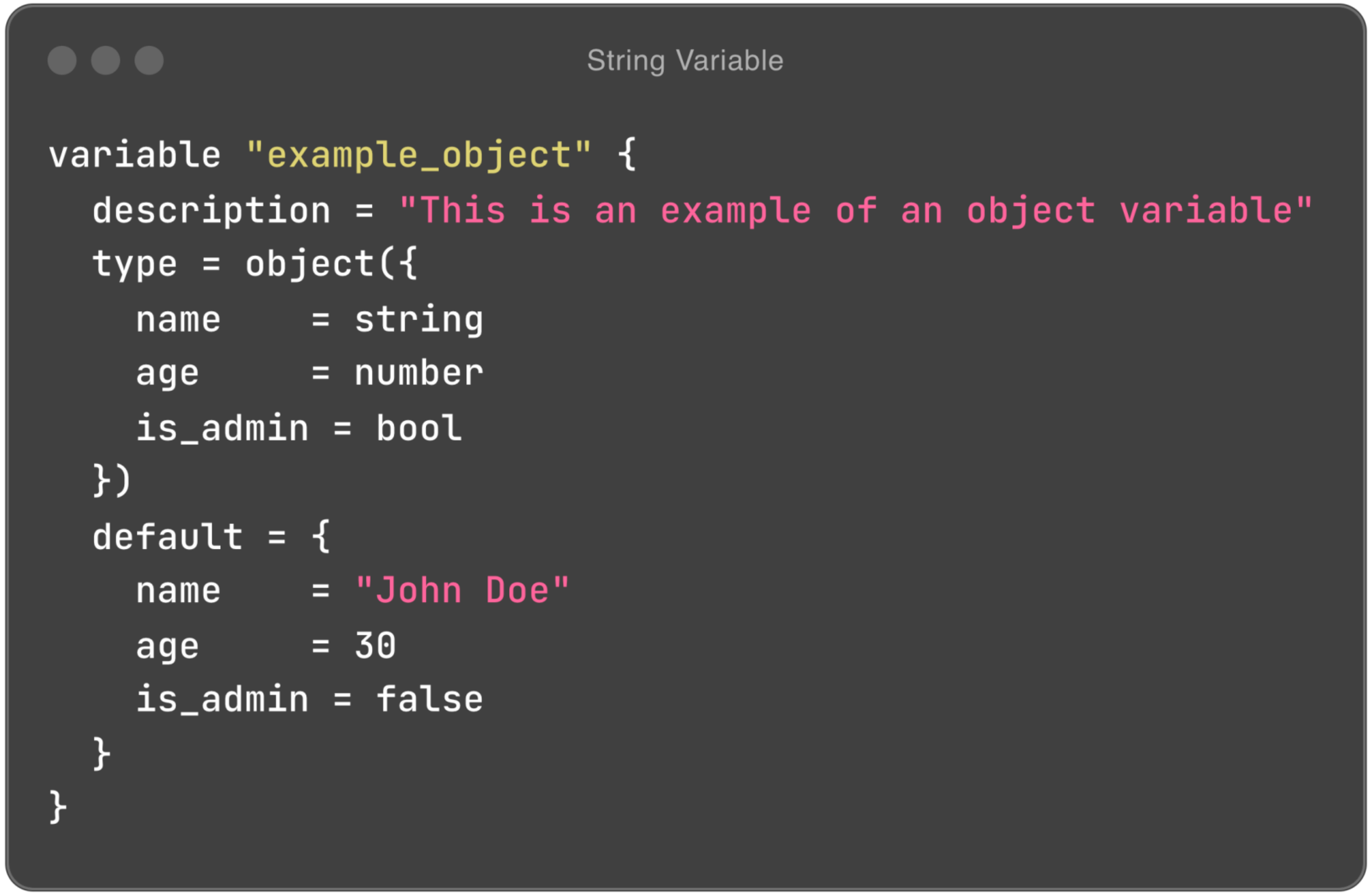
```
variable "example_string" {
  description = "This is an example of a string variable"
  type        = string
  default     = "Hello, Terraform!"
}

variable "example_number" {
  description = "This is an example of a number variable"
  type        = number
  default     = 42
}

variable "example_boolean" {
  description = "This is an example of a boolean variable"
  type        = bool
  default     = true
}
```

Input Variables: List and Map



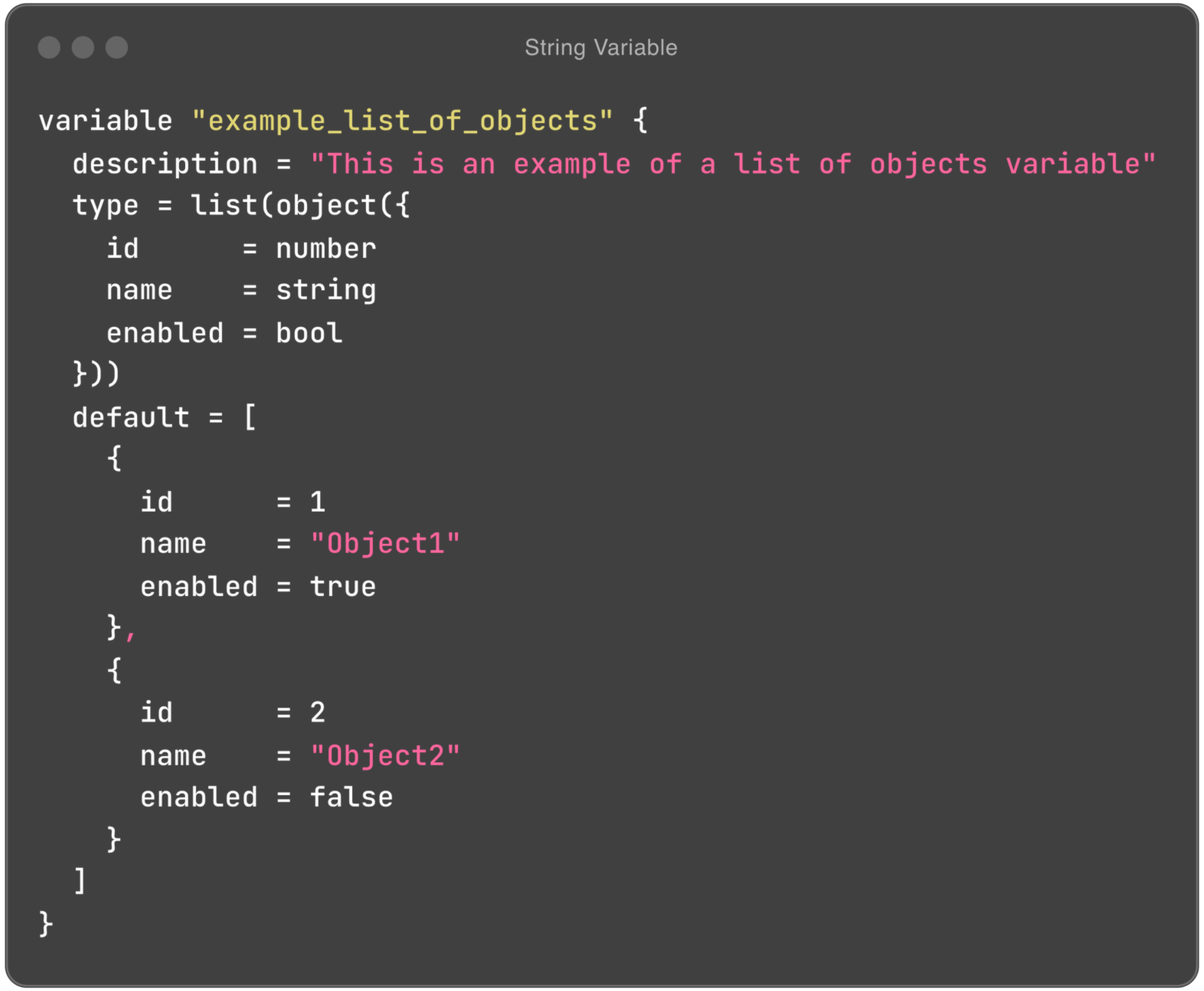
```
variable "example_list" {
  description = "This is an example of a list variable"
  type        = list(string)
  default     = ["apple", "banana", "cherry"]
}

variable "example_map" {
  description = "This is an example of a map variable"
  type        = map(string)
  default     = {
    "key1" = "value1"
    "key2" = "value2"
    "key3" = "value3"
  }
}
```

Input Variables: Object



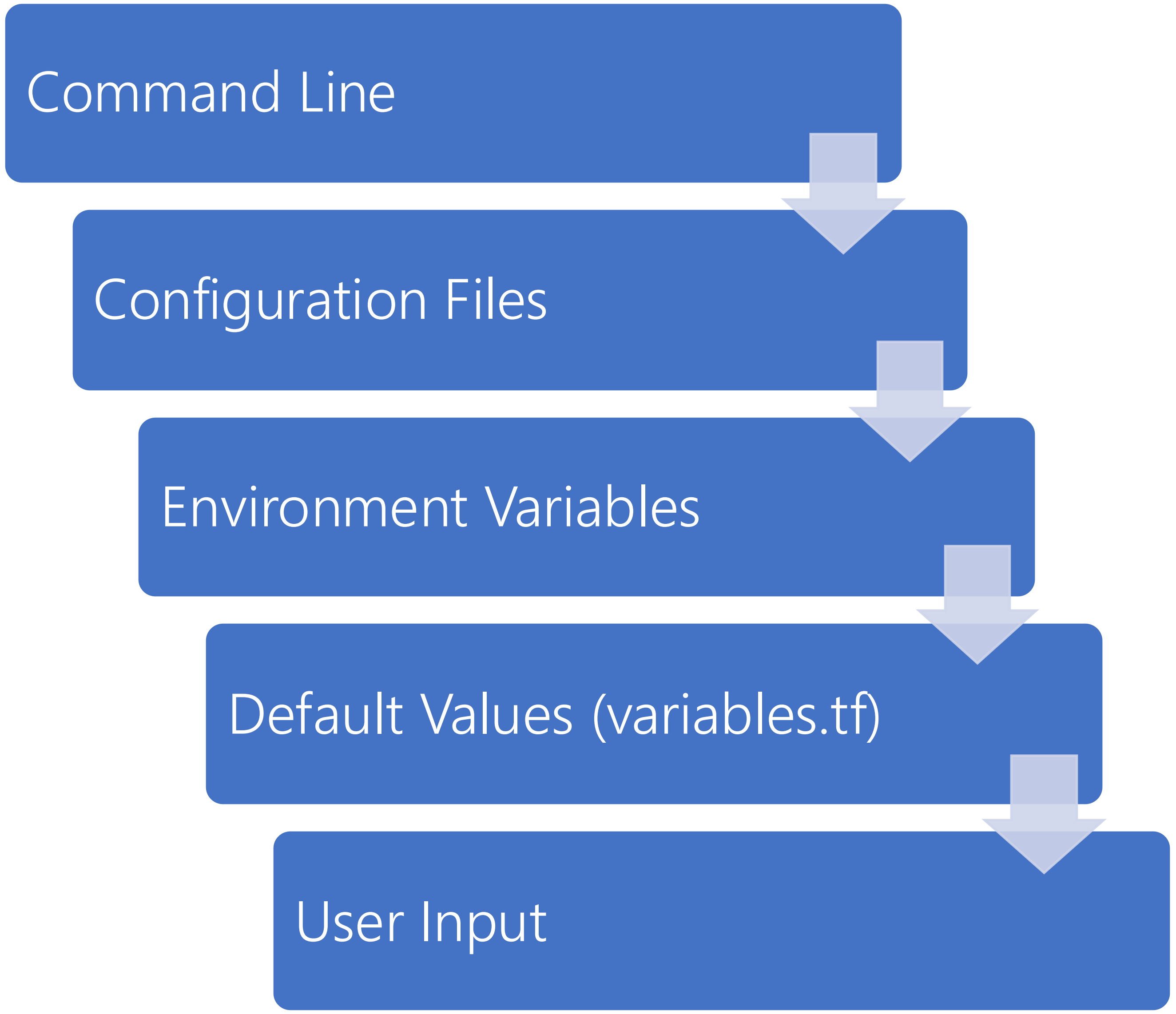
```
variable "example_object" {
  description = "This is an example of an object variable"
  type = object({
    name     = string
    age      = number
    is_admin = bool
  })
  default = {
    name     = "John Doe"
    age      = 30
    is_admin = false
  }
}
```

Input Variables: List of Object


String Variable

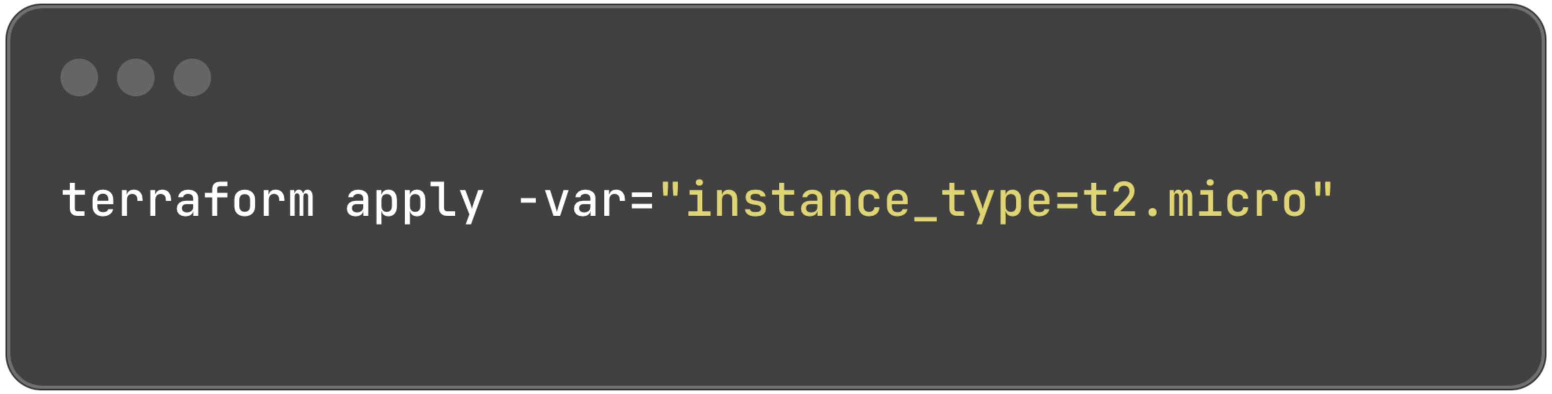
```
variable "example_list_of_objects" {
  description = "This is an example of a list of objects variable"
  type = list(object({
    id      = number
    name    = string
    enabled = bool
  }))
  default = [
    {
      id      = 1
      name    = "Object1"
      enabled = true
    },
    {
      id      = 2
      name    = "Object2"
      enabled = false
    }
  ]
}
```

How to Set Variables



How to Set Variables: Command Line

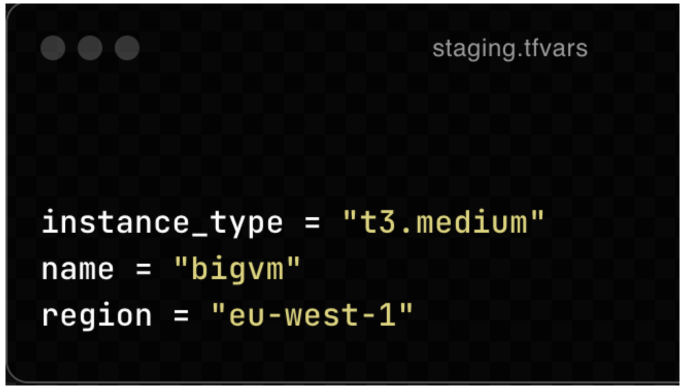
- Variables can be set directly using the **-var** flag in Terraform commands
- Command-line flags have the highest precedence



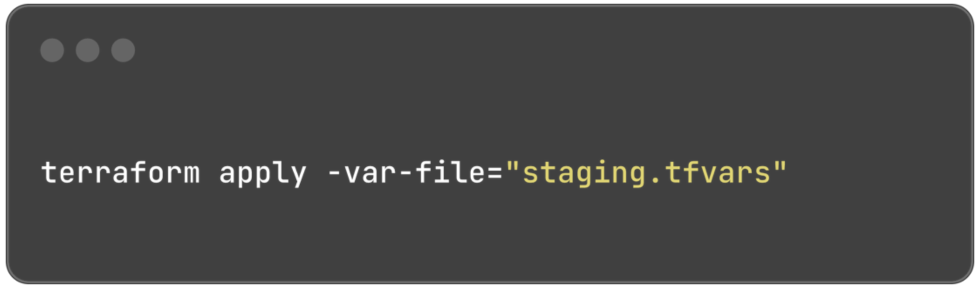
```
terraform apply -var="instance_type=t2.micro"
```

How to Set Variables: Configuration Files

- Variables can be defined in ***.tfvars** or ***.auto.tfvars** files
- Terraform automatically loads **.auto.tfvars** or **terraform.tfvars** files, while others must be explicitly specified.



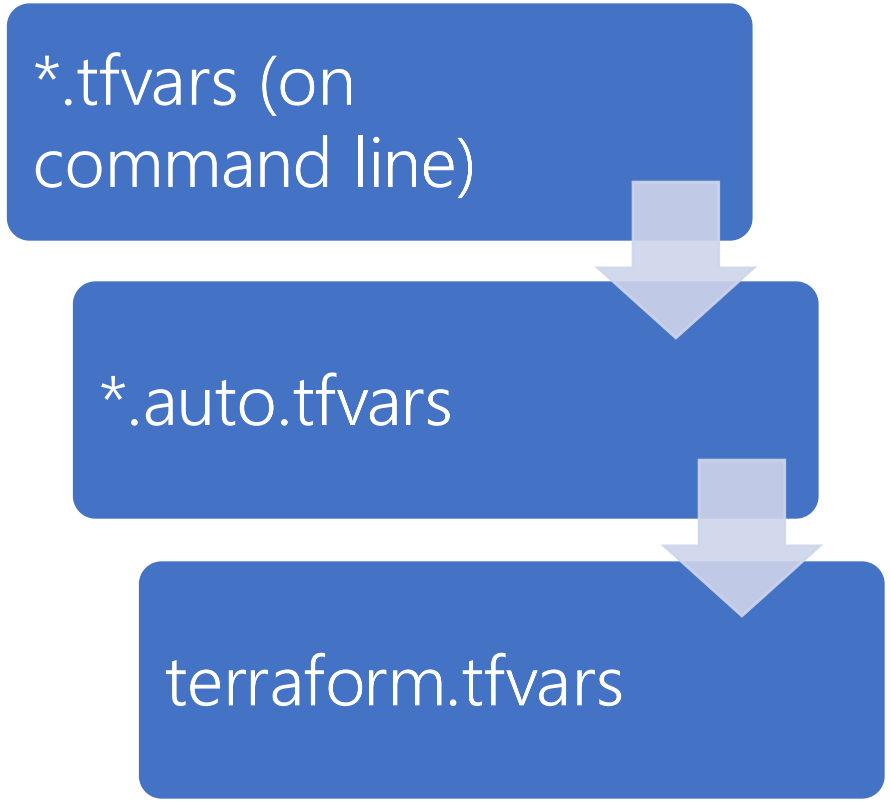
```
instance_type = "t3.medium"
name = "bigvm"
region = "eu-west-1"
```



```
terraform apply -var-file="staging.tfvars"
```

How to Set Variables: Configuration Files

- There is a precedence between configuration files

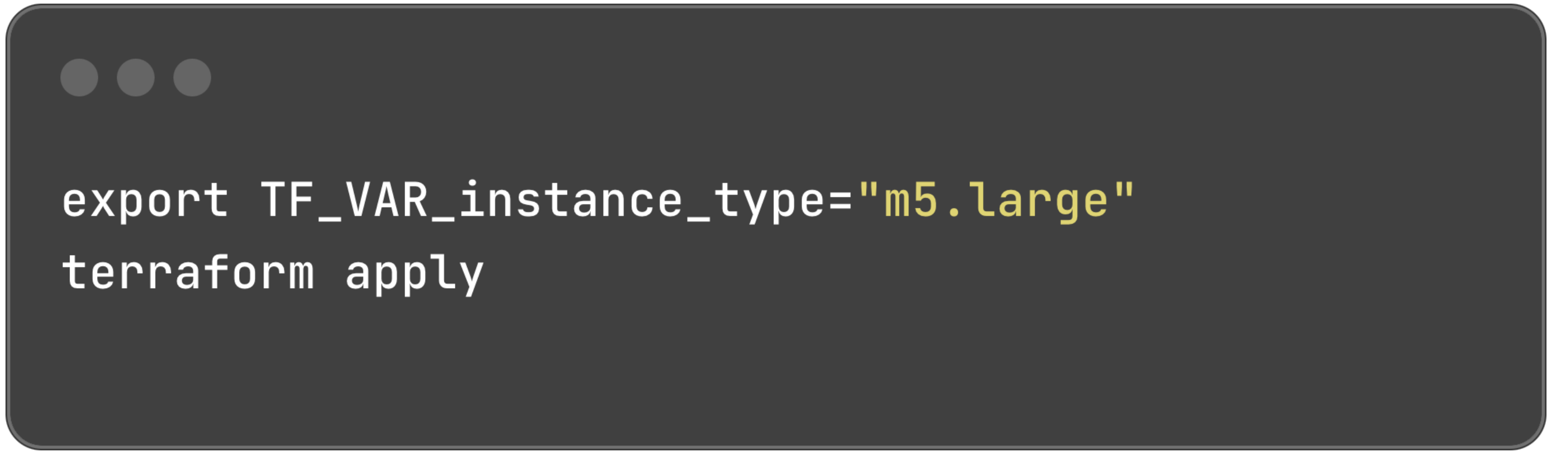


How to Set Variables: Why several config files?

- **.auto.tfvars**: Store default or shared values across all environments (e.g., general settings that should always apply).
- **.tfvars**: Store environment-specific configurations (e.g., different values for production, staging, and development) and apply them explicitly making them preferred

How to Set Variables: Env vars

- Environment variables are prefixed with TF_VAR_ followed by the variable name.



```
export TF_VAR_instance_type="m5.large"
terraform apply
```

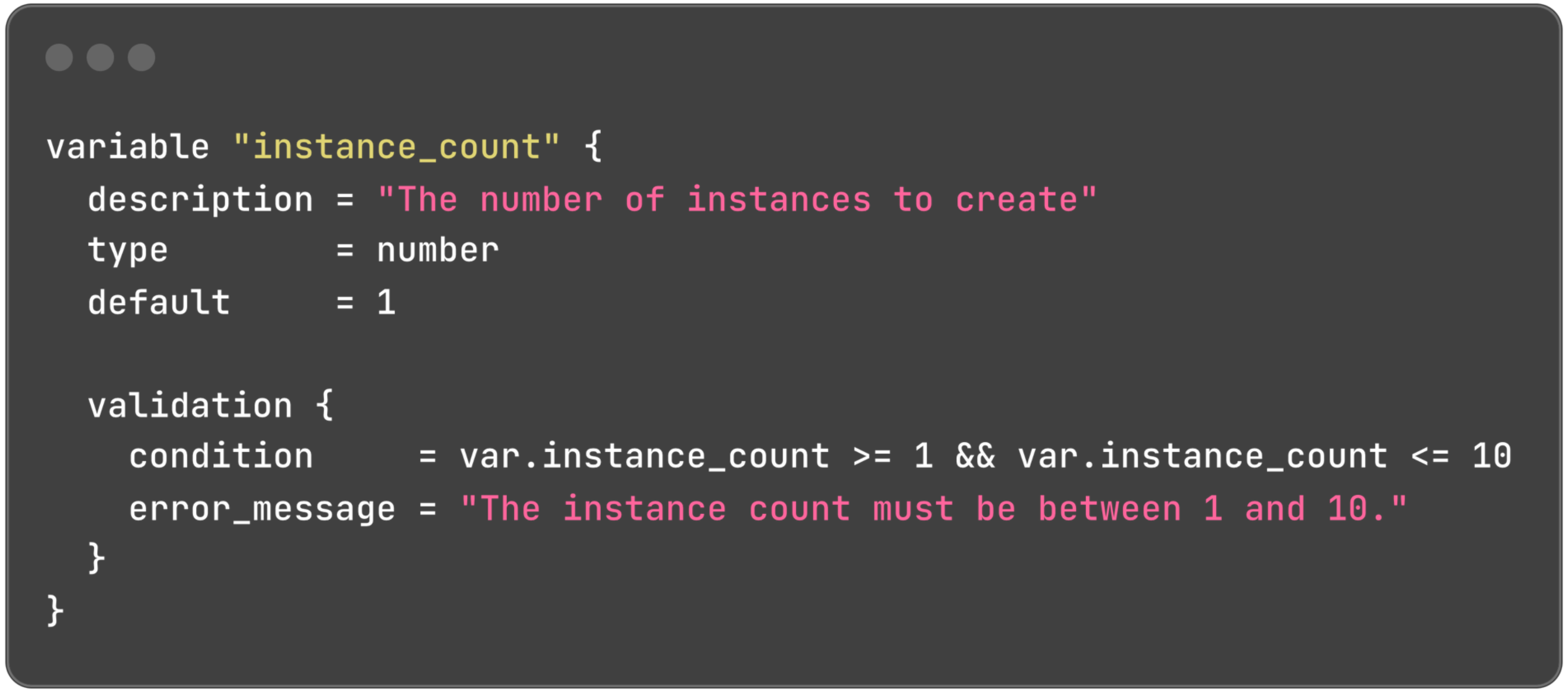
Input Variables Validation

- Variable validation in Terraform allows you to define custom conditions that input variable values must meet
- This ensures that only valid data is provided, helping to catch errors early.
- Validation is added inside a variable block using the **validation** argument, where you define a condition (**condition**) and an error message (**error_message**).

Input Variables Validation

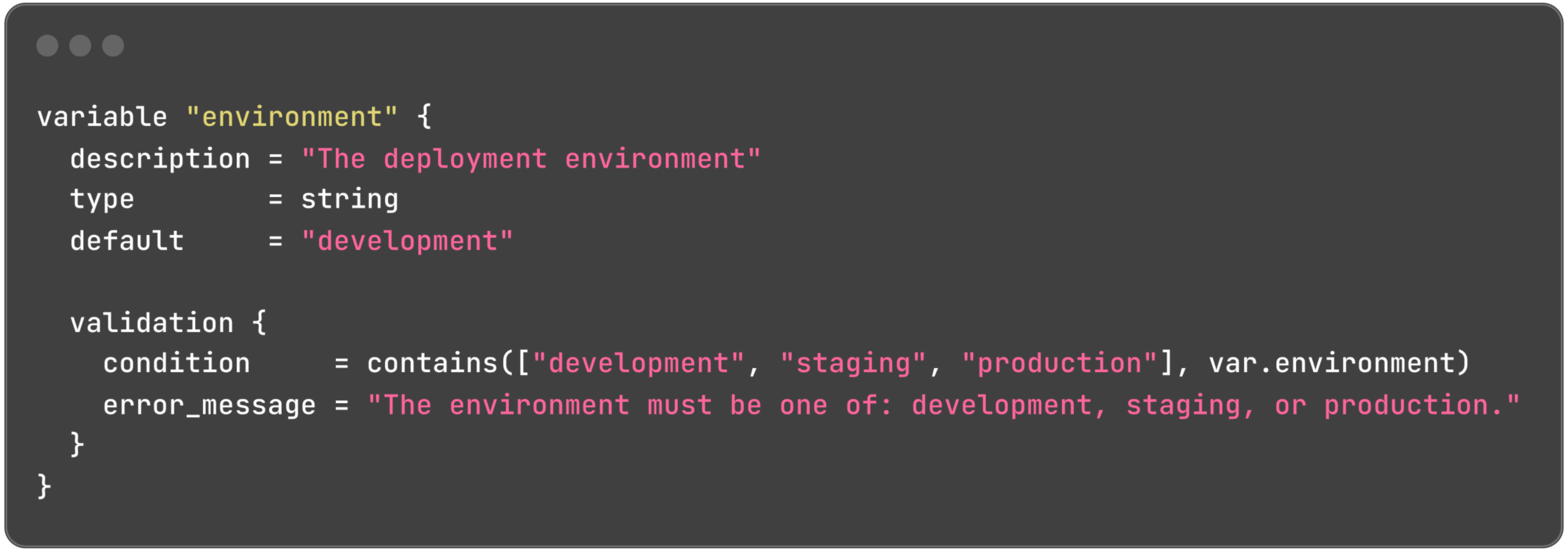
- **condition**: A logical expression that must evaluate to **true** for the variable to be valid.
- **error_message**: A custom error message displayed when the validation condition is not met.

Input Variables Validation



```
variable "instance_count" {
  description = "The number of instances to create"
  type        = number
  default     = 1

  validation {
    condition     = var.instance_count >= 1 && var.instance_count <= 10
    error_message = "The instance count must be between 1 and 10."
  }
}
```

Input Variables Validation



```
variable "environment" {
  description = "The deployment environment"
  type        = string
  default     = "development"

  validation {
    condition     = contains(["development", "staging", "production"], var.environment)
    error_message = "The environment must be one of: development, staging, or production."
  }
}
```

Local Variables

Terraform

Local Variables

- Variables that are local to the code
- They are defined, assigned, and used in the same block of code
- They are defined in the “locals” block
- Local variables can be declared once and used any number of times
- These can be accessed as objects by using the format of **local.Variable_Name**

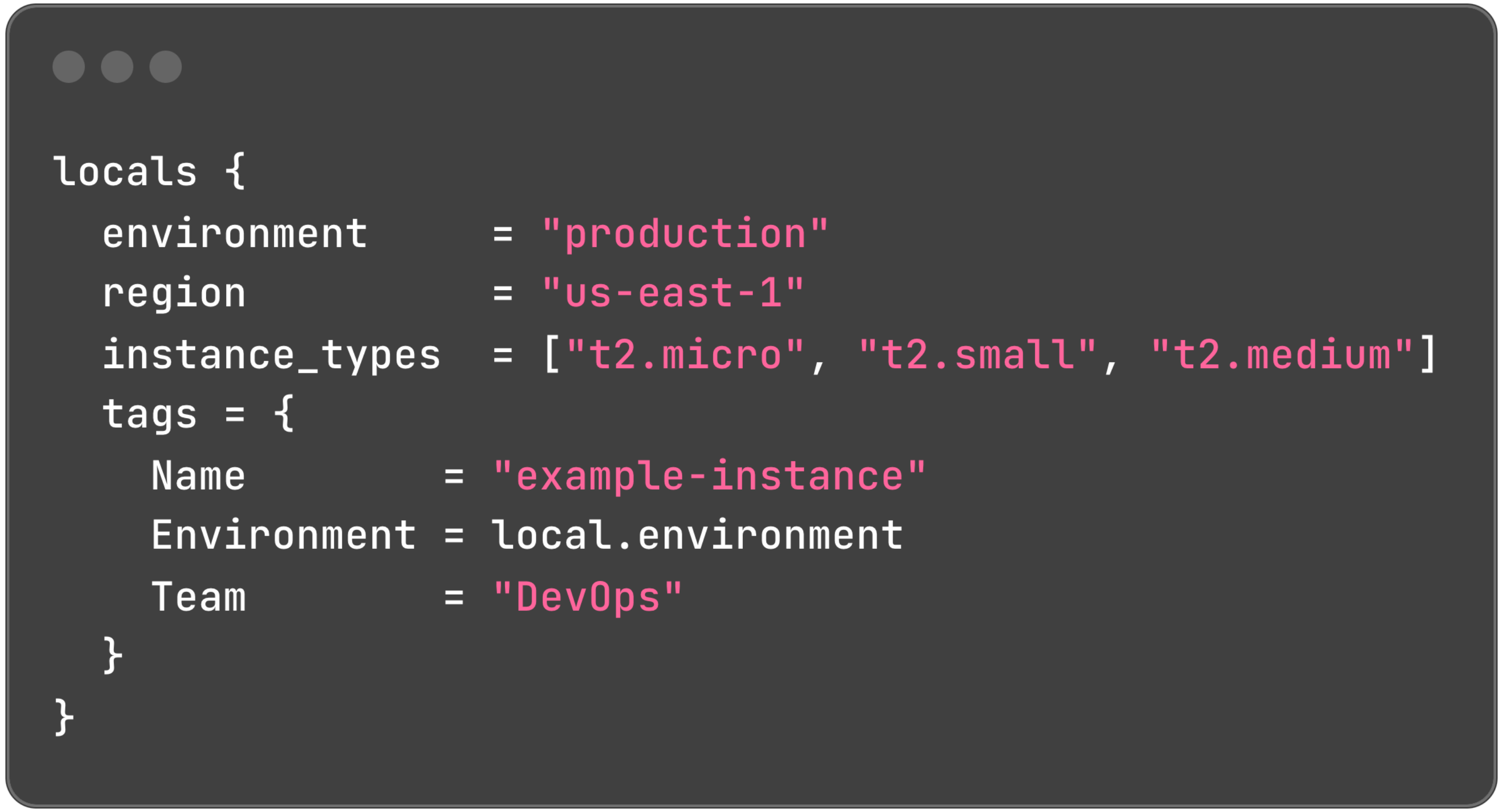
Local Variables

- Local values can be helpful to avoid repeating the same values or expressions multiple
- times in a configuration
- The ability to easily change the value in a central place is the key advantage of local values
- Each **locals** block can have as many locals as needed, and there can be any number of locals blocks
- The expression of a local value can refer to other locals, but as usual reference cycles are not allowed

Local Variables: Differences for inputs

- Unlike variables found in programming languages, Terraform locals don't change values
- Unlike input variables, locals are not set directly by users of your configuration

Local Variables



```hcl
locals {
  environment    = "production"
  region         = "us-east-1"
  instance_types = ["t2.micro", "t2.small", "t2.medium"]
  tags = {
    Name        = "example-instance"
    Environment = local.environment
    Team        = "DevOps"
  }
}
```

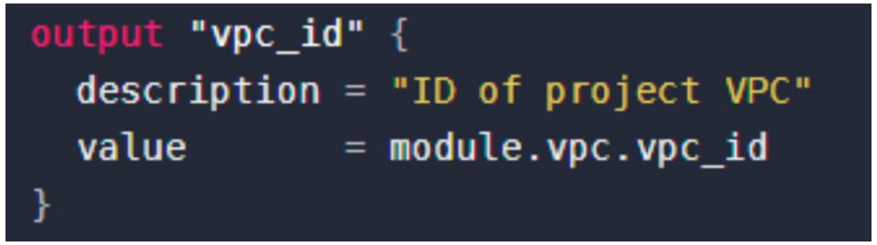
Outputs

Terraform

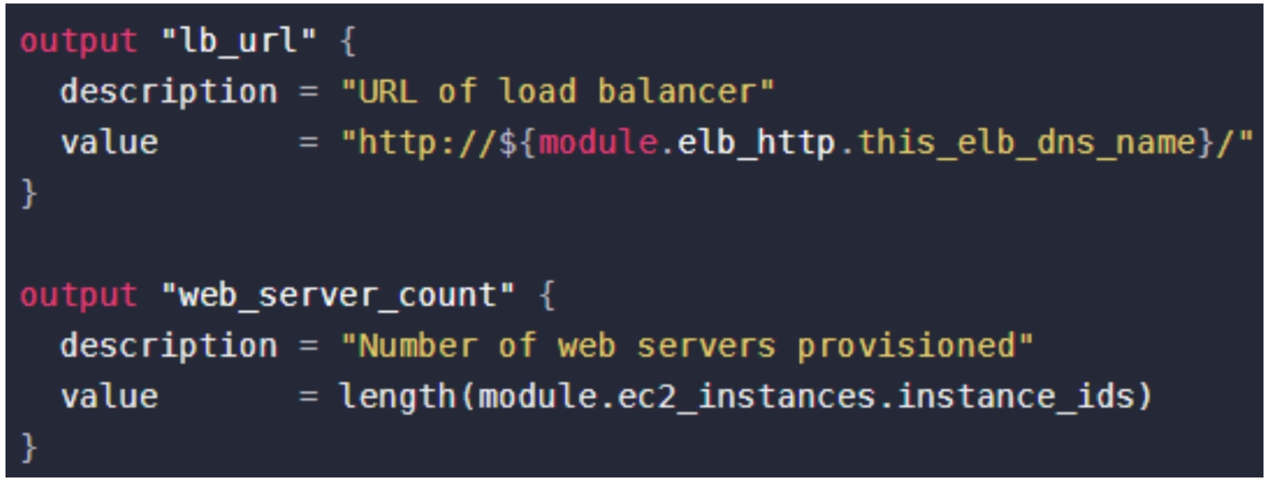
Outputs

- Terraform will store hundreds or even thousands of attribute values for all the defined resources in state file
- Outputs can be used for user reference, but it can also act as an input to other resources being created via Terraform
- We can use output variables to organize data to be easily queried and shown back to the Terraform user
- Each output value exported by a module must be declared using an output block

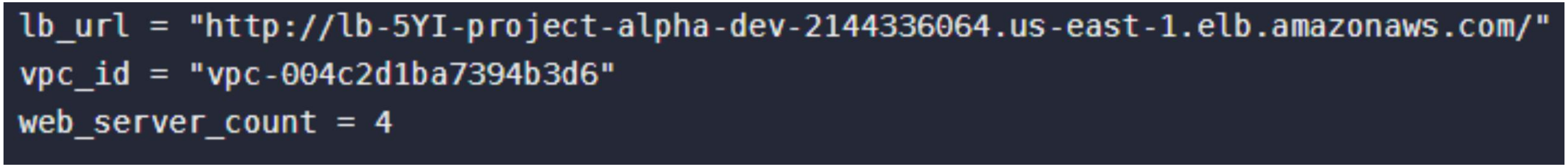
Outputs



```
output "vpc_id" {
  description = "ID of project VPC"
  value       = module.vpc.vpc_id
}
```



```
output "lb_url" {
  description = "URL of load balancer"
  value       = "http://${module.elb_http.this_elb_dns_name}/"
}

output "web_server_count" {
  description = "Number of web servers provisioned"
  value       = length(module.ec2_instances.instance_ids)
}
```



```
lb_url = "http://lb-5YI-project-alpha-dev-2144336064.us-east-1.elb.amazonaws.com/"
vpc_id = "vpc-004c2d1ba7394b3d6"
web_server_count = 4
```

Interpolation

Terraform

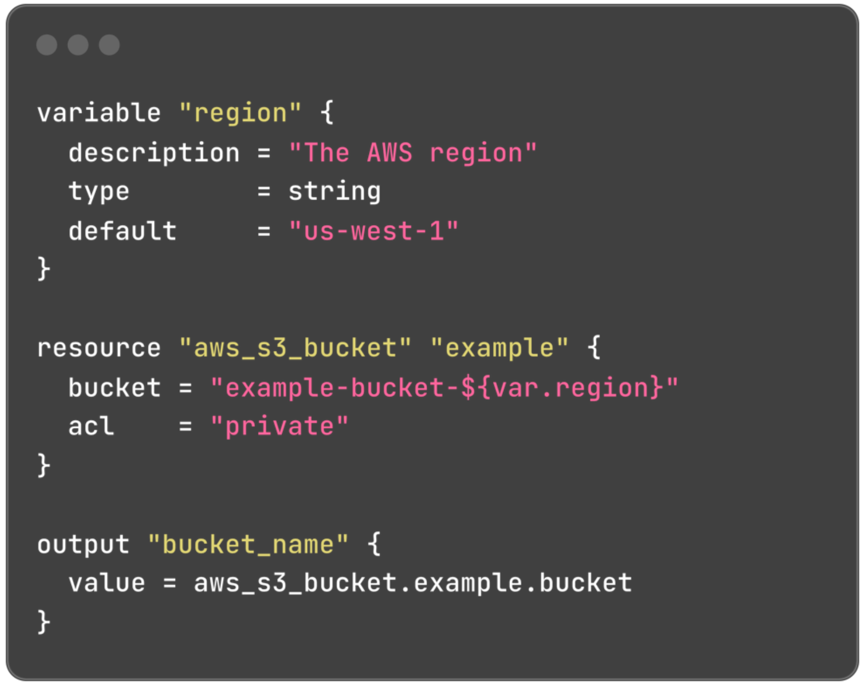
Direct access to variables

- When you want to take the direct value of any kind of variable, you use the direct mode to instantiate

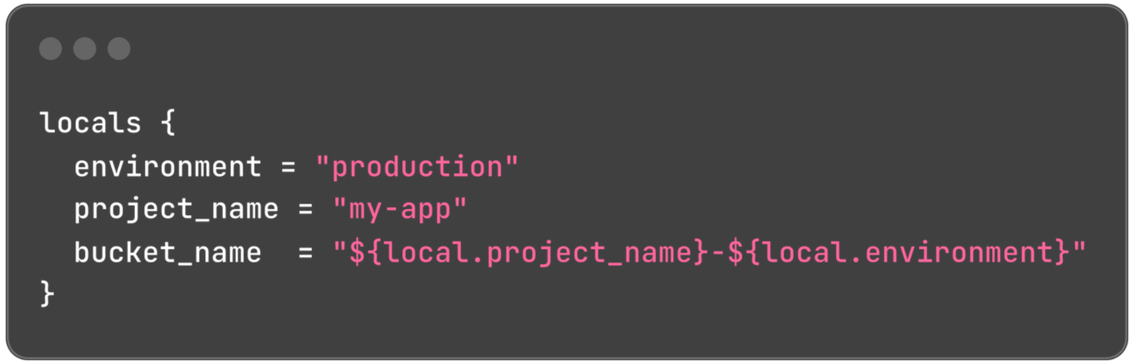
```
resource "azurerm_virtual_machine" "example" {
  name = var.name
  location = var.location
  size = local.vm_size
}
```

Interpolation

- In Terraform, you can interpolate other values, using **${...}**
- You can use simple math functions, refer to other variables, or use conditionals (if-else)



```
variable "region" {
  description = "The AWS region"
  type        = string
  default     = "us-west-1"
}

resource "aws_s3_bucket" "example" {
  bucket = "example-bucket-${var.region}"
  acl    = "private"
}

output "bucket_name" {
  value = aws_s3_bucket.example.bucket
}
```



```
locals {
  environment = "production"
  project_name = "my-app"
  bucket_name  = "${local.project_name}-${local.environment}"
}
```

Conditions

Terraform

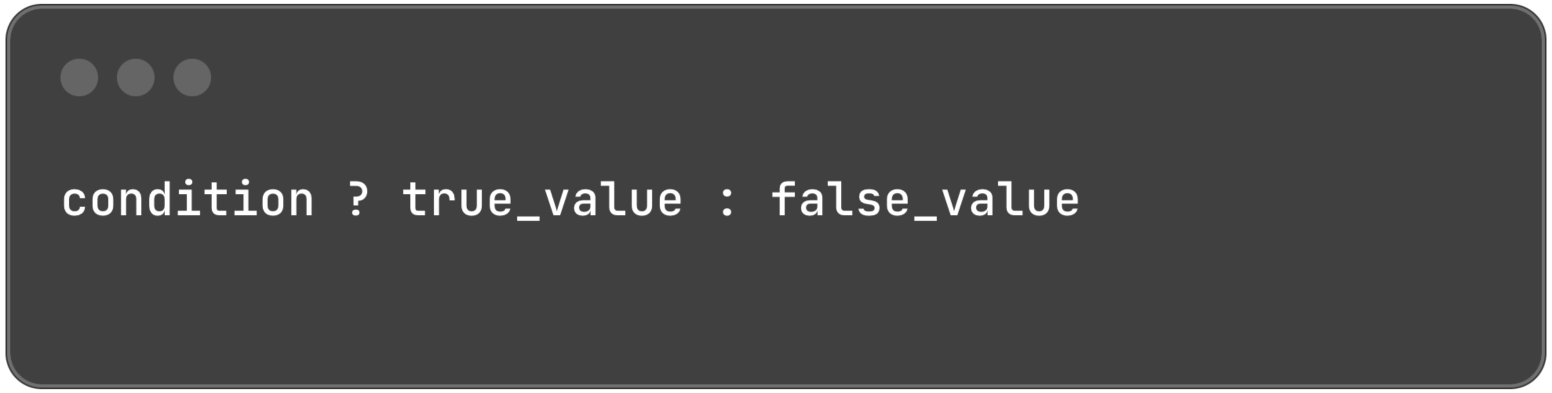
Conditionals

- Conditionals in Terraform allow you to make decisions and choose between values based on a condition. This is achieved using the ternary operator:

```
condition ? true_value : false_value
```

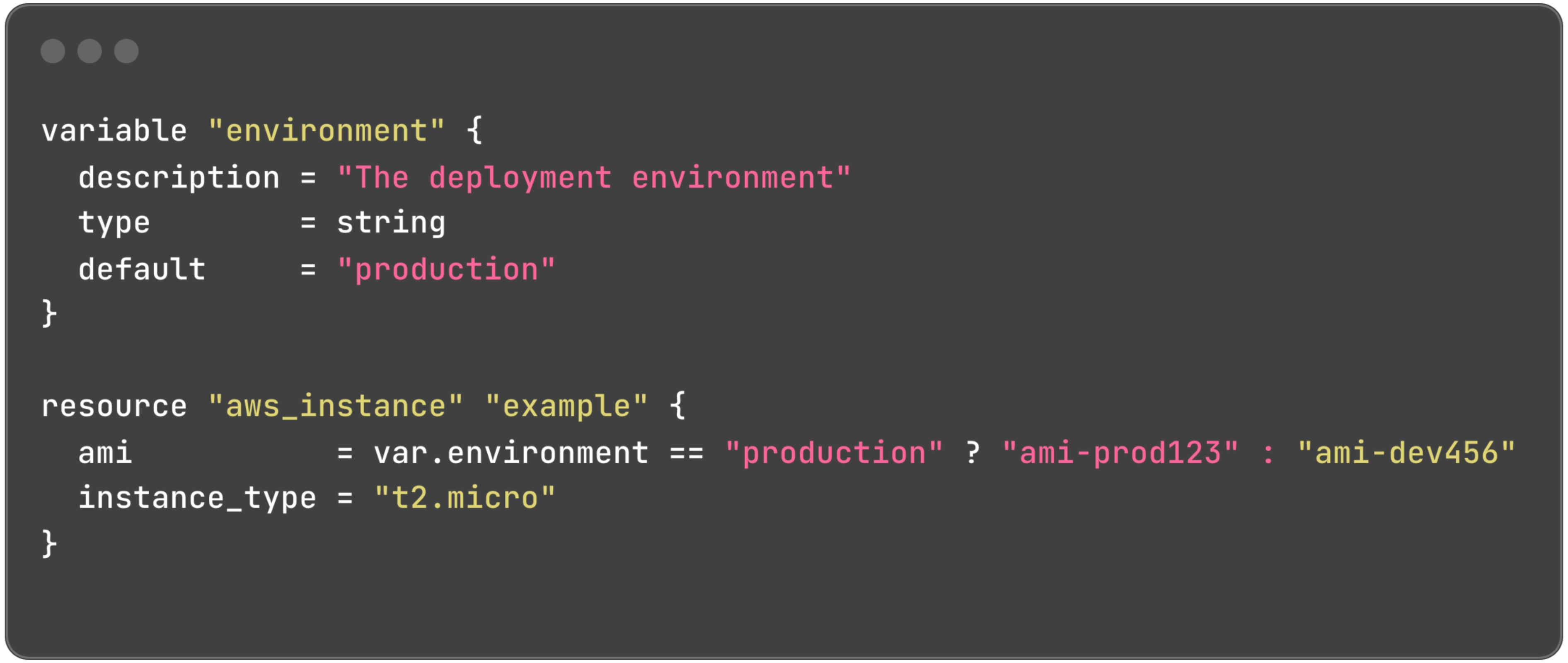
Conditionals: How it works

- **condition**: An expression that evaluates to true or false
- **true_value**: The value returned if the condition is true
- **false_value**: The value returned if the condition is false.



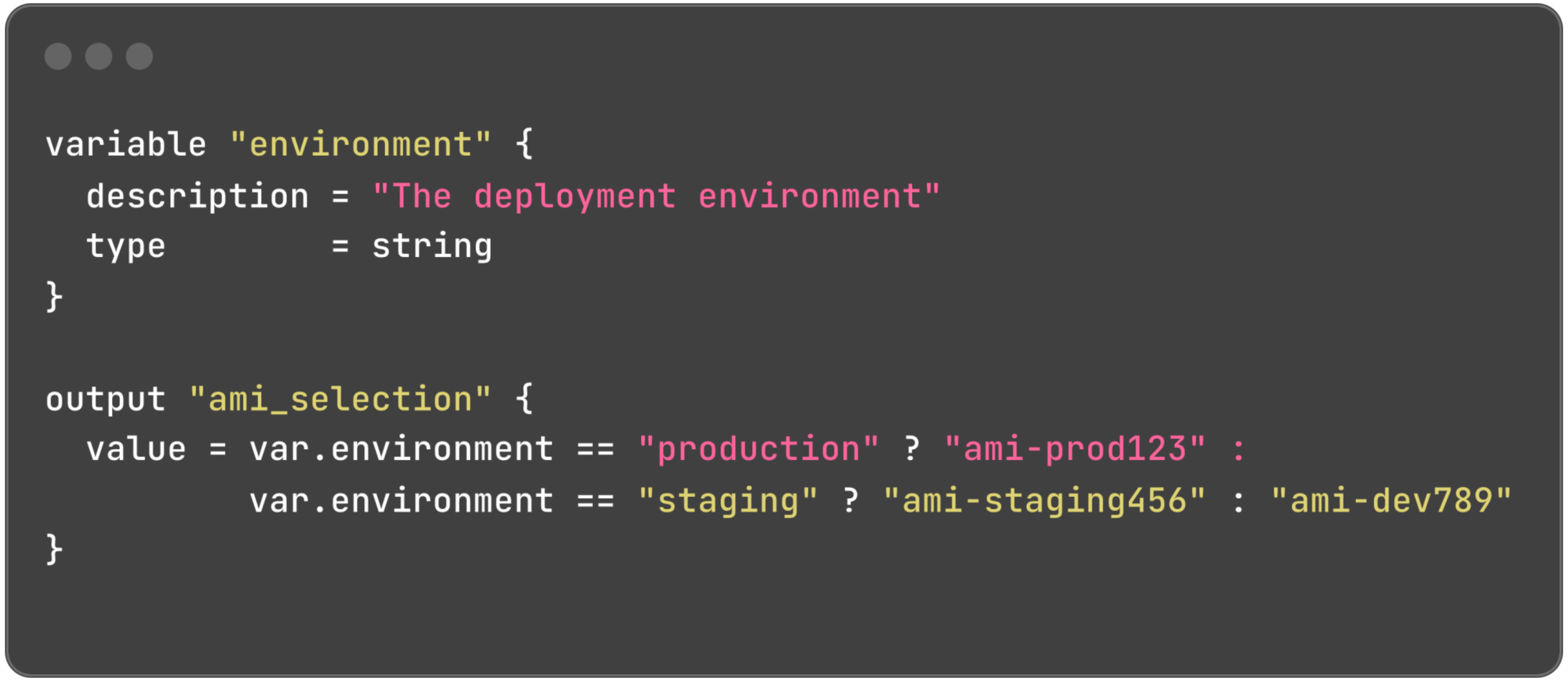
```
condition ? true_value : false_value
```

Conditionals: Single Conditions



```
variable "environment" {
  description = "The deployment environment"
  type        = string
  default     = "production"
}

resource "aws_instance" "example" {
  ami           = var.environment == "production" ? "ami-prod123" : "ami-dev456"
  instance_type = "t2.micro"
}
```

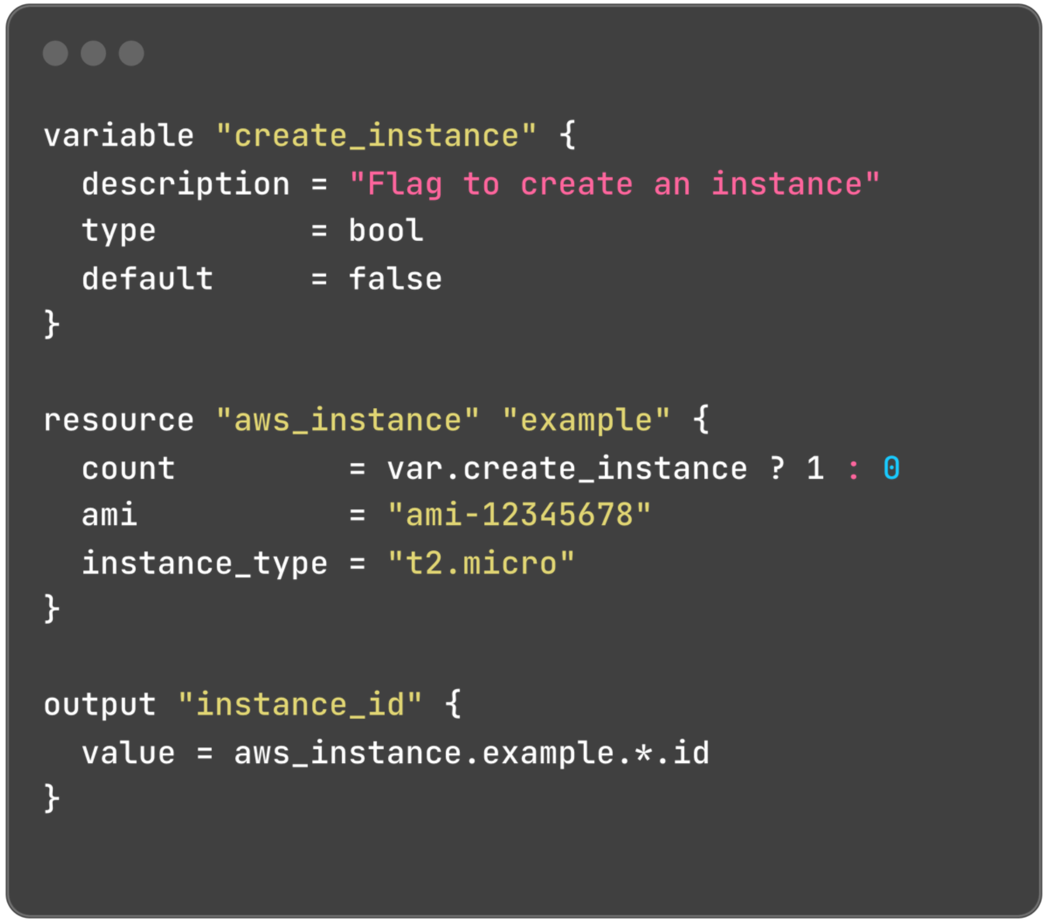
Conditionals: Nested Conditions



```
variable "environment" {
  description = "The deployment environment"
  type        = string
}

output "ami_selection" {
  value = var.environment == "production" ? "ami-prod123" :
          var.environment == "staging" ? "ami-staging456" : "ami-dev789"
}
```

Conditionals: Count

- The **count** property in Terraform is used to create multiple instances of a resource or to conditionally create or omit resources
- By default, all resources have the **count** property set to 1
- If **count** is equal to 0 the resource is not created



```
variable "create_instance" {
  description = "Flag to create an instance"
  type        = bool
  default     = false
}

resource "aws_instance" "example" {
  count         = var.create_instance ? 1 : 0
  ami           = "ami-12345678"
  instance_type = "t2.micro"
}

output "instance_id" {
  value = aws_instance.example.*.id
}
```

Conditionals: Key Benefits

- Conditional Resource Creation: Dynamically create or skip resources based on conditions.
- Dynamic Scaling: Scale resources (e.g., subnets or instances) based on input lists or numbers.

Conditionals: Best Practices

- Clarity: Use descriptive variable names for conditions (e.g., create_instance)
- Avoid Complex Nesting: Avoid deeply nested conditionals to maintain readability

Iterations

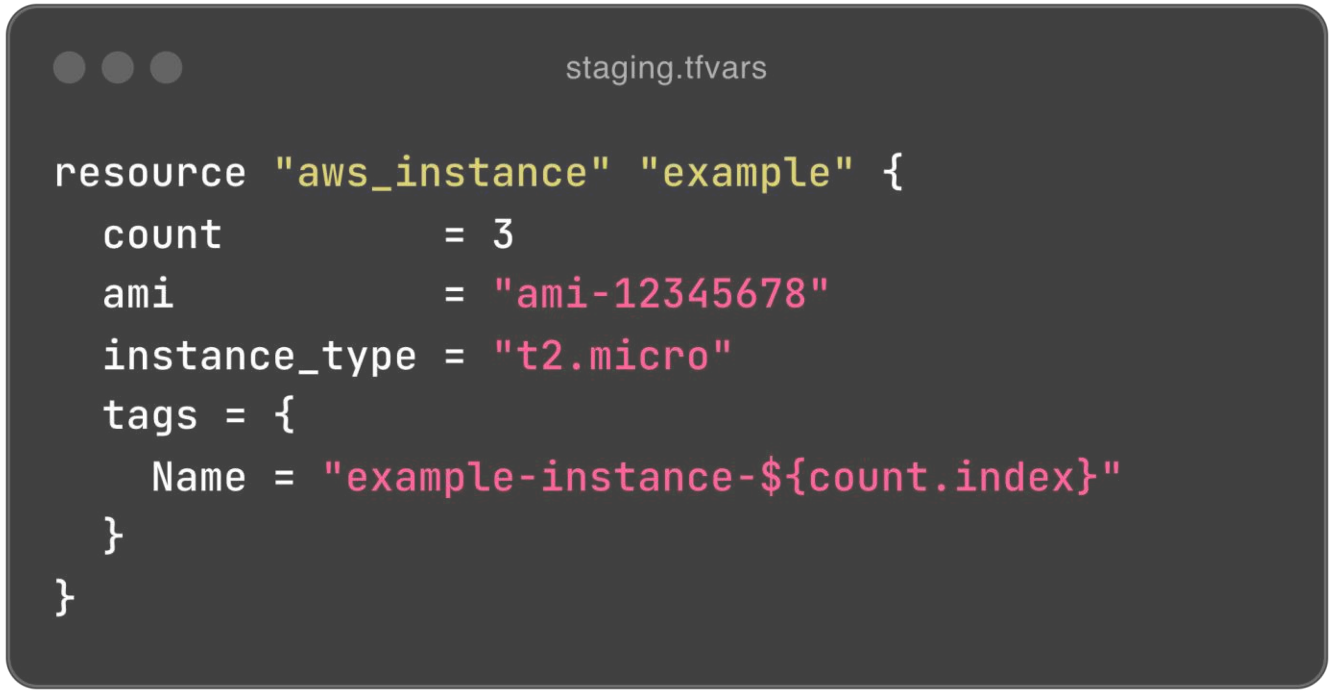
Terraform

Iterations

- Terraform supports iterations through constructs like **count**, **for_each**, and the **for** expression
- These are essential for dynamically creating resources, looping through lists, or building complex expressions.

Iterations: **count**

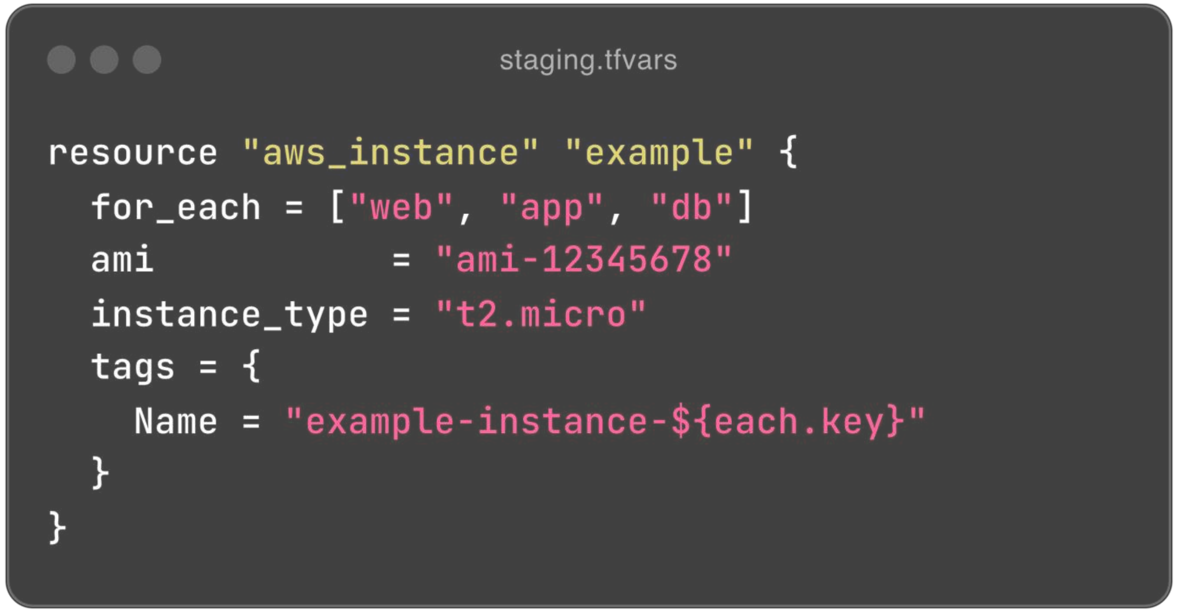
- The **count** meta-argument is used to create multiple instances of a resource
- It works by specifying the number of instances (count) to create, and each instance gets an index starting from 0.


staging.tfvars

```
resource "aws_instance" "example" {
  count         = 3
  ami           = "ami-12345678"
  instance_type = "t2.micro"
  tags = {
    Name = "example-instance-${count.index}"
  }
}
```

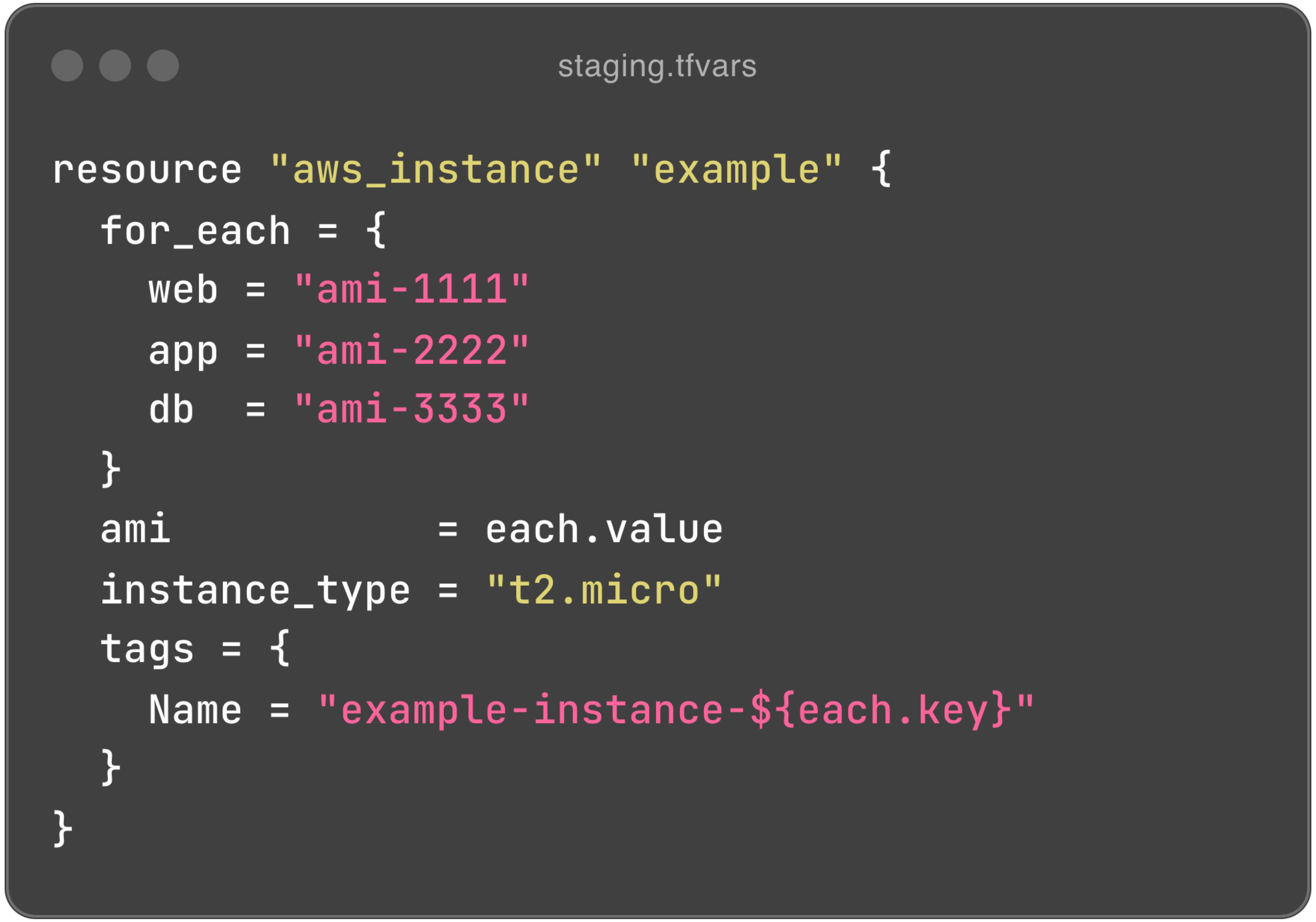
Iterations: **for_each**

- The **for_each** meta-argument allows you to loop over a collection (e.g., a list or map)
- It is used when you want to iterate over specific items and provides better flexibility than count.


staging.tfvars

```
resource "aws_instance" "example" {
  for_each = ["web", "app", "db"]
  ami           = "ami-12345678"
  instance_type = "t2.micro"
  tags = {
    Name = "example-instance-${each.key}"
  }
}
```

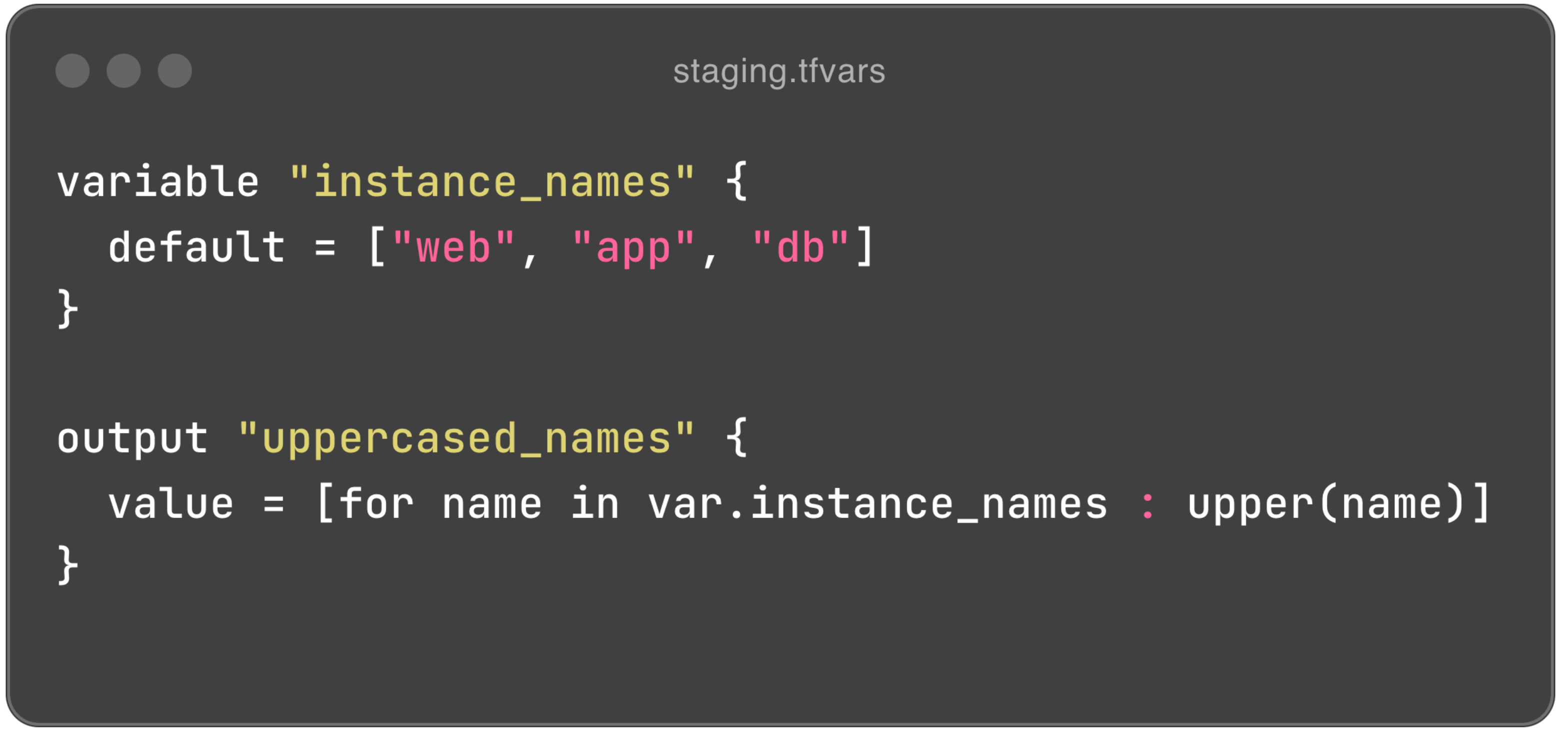
Iterations: **for_each**



```
resource "aws_instance" "example" {
  for_each = {
    web = "ami-1111"
    app = "ami-2222"
    db  = "ami-3333"
  }
  ami           = each.value
  instance_type = "t2.micro"
  tags = {
    Name = "example-instance-${each.key}"
  }
}
```

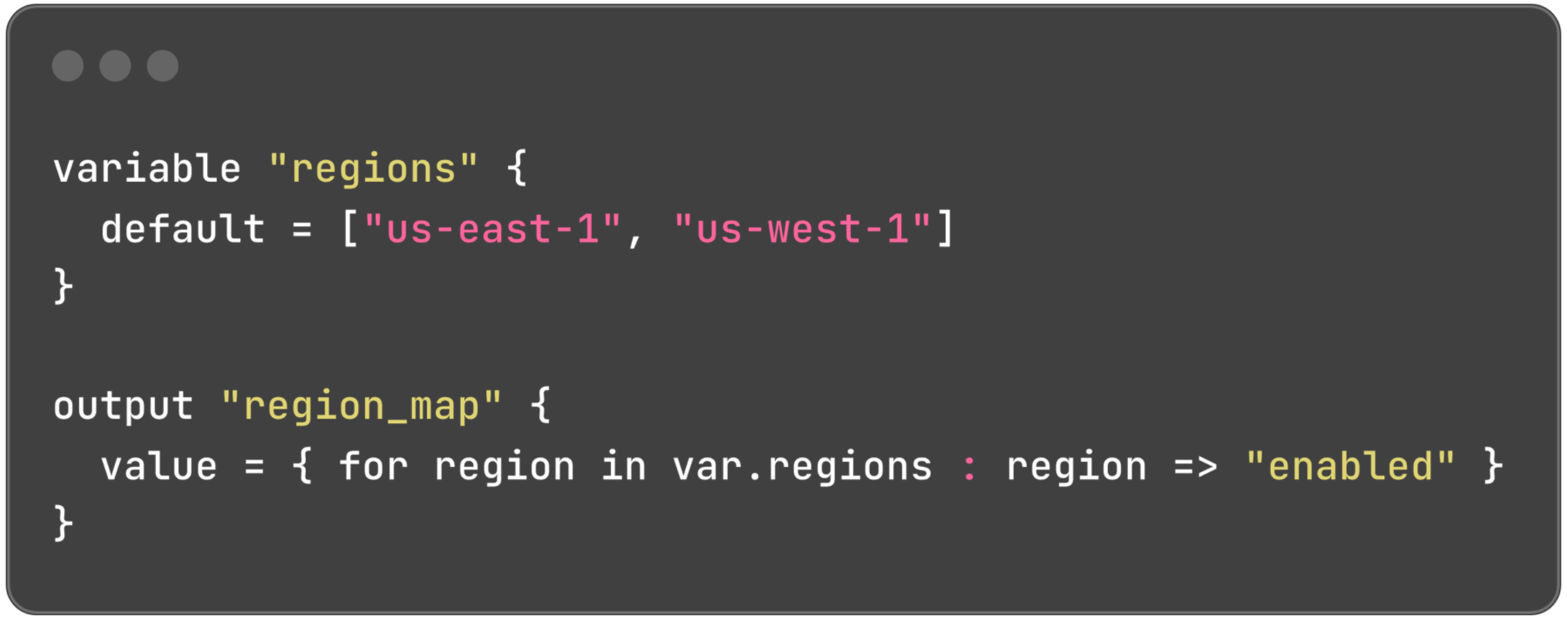
Iterations

- The **for** expression is used to transform or filter collections. You can use it to create new lists or maps dynamically.

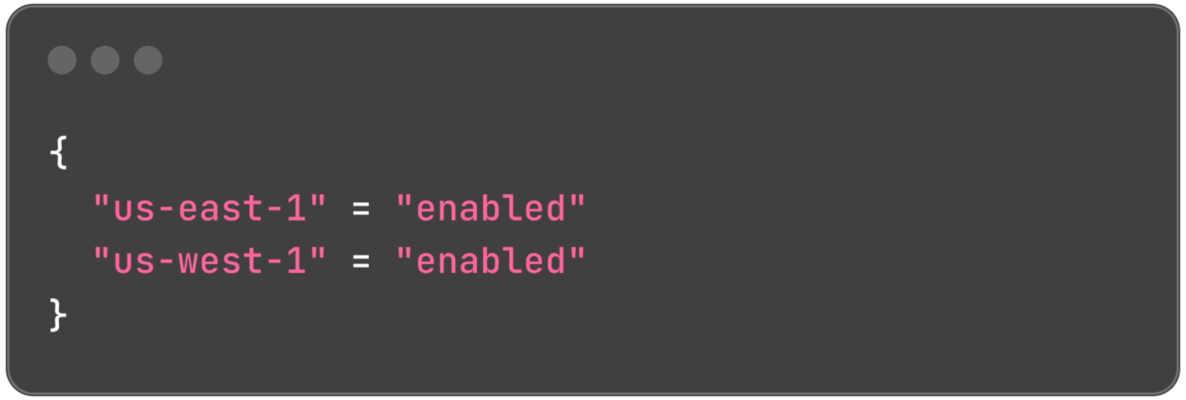


```
variable "instance_names" {
  default = ["web", "app", "db"]
}

output "uppercased_names" {
  value = [for name in var.instance_names : upper(name)]
}
```

Iterations



```
variable "regions" {
  default = ["us-east-1", "us-west-1"]
}

output "region_map" {
  value = { for region in var.regions : region => "enabled" }
}
```



```
{
  "us-east-1" = "enabled"
  "us-west-1" = "enabled"
}
```

Iterations

- By using these iteration tools, Terraform allows dynamic and scalable resource creation while keeping your configuration DRY (Don't Repeat Yourself).

Feature	Use Case
count	When you need to create multiple identical resources.
for_each	When you need to iterate over specific elements (lists/maps).
for	When transforming, filtering, or dynamically building lists or maps.

Data Sources

Terraform

Data Sources

- Data sources in Terraform allow you to query existing resources or external information and use it in your configuration
- They are particularly useful when you want to reference existing resources instead of creating new ones or fetch dynamic information from providers (e.g., AMI IDs, VPC IDs, or environment-specific values).

Data Sources: Why?

- Reuse Existing Resources: Avoid duplicating infrastructure already managed outside your Terraform configuration
- Dynamic Updates: Always fetch the latest state of resources, such as the newest AMI or updated VPC information.
- Environment-Specific Configuration: Query resources based on environment-specific tags or attributes.
- Integration with External Systems: Data sources enable integration with systems like AWS, Terraform Cloud, external files, or third-party APIs.

Data Sources: Syntax

- **provider_resource_type**: The resource type you are querying.
- **name**: A unique name to reference the data source.
- Arguments: Used to filter or query specific data.

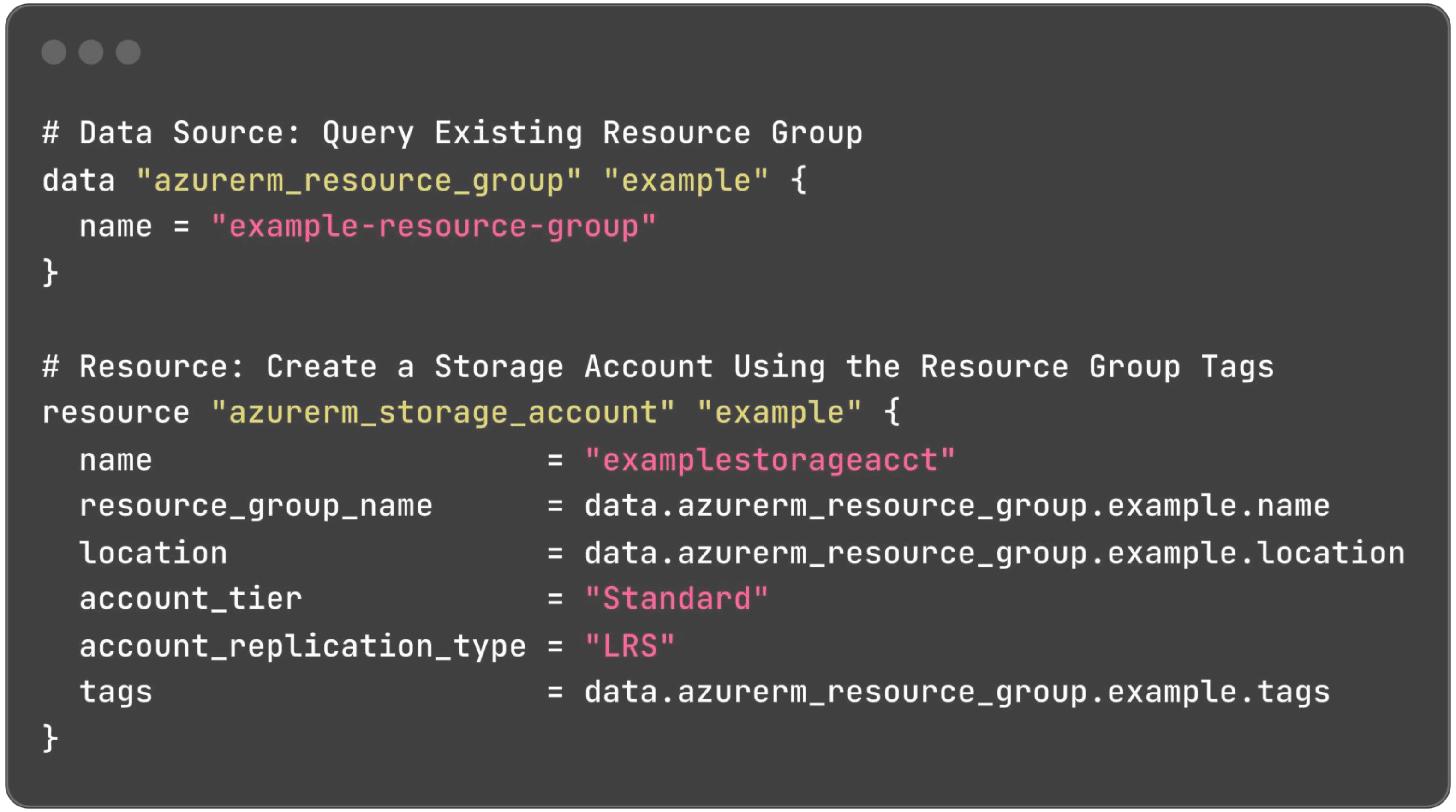
```
data "provider_resource_type" "name" {
  # Arguments to query the resource
}
```

Data Sources: Syntax

- To reference you use the same concept as resources

```
data.provider_resource_type.name
```

Data Sources: Example



```hcl
# Data Source: Query Existing Resource Group
data "azurerm_resource_group" "example" {
  name = "example-resource-group"
}

# Resource: Create a Storage Account Using the Resource Group Tags
resource "azurerm_storage_account" "example" {
  name                     = "examplestorageacct"
  resource_group_name      = data.azurerm_resource_group.example.name
  location                 = data.azurerm_resource_group.example.location
  account_tier             = "Standard"
  account_replication_type = "LRS"
  tags                     = data.azurerm_resource_group.example.tags
}
```

Data Sources: Key Points to Remember

- Data sources only fetch data; they do not create or manage resources
- Combine data sources with resources or outputs for dynamic and reusable configurations
- Data sources execute queries during the planning phase (**terraform plan**).

Data Sources: Best Practices

- Use filters to precisely define the data you need (e.g., by tags, names, or other identifiers)
- Limit usage of data sources to avoid fetching unnecessary data, which can slow down execution
- Document data source usage to ensure clarity for future updates

Demo – Your first Terraform Code

Terraform

Lab 02 – Add variables to your code

Terraform