Module 3: Advanced Terraform Functions

Terraform-Intermediate-Day1-Module3

Learning Objectives

By the end of this module we will understand:

- Format strings for consistent naming
- Generate dynamic configuration files
- Combine multiple maps into one
- Build lookup tables from lists
- Flatten complex nested structures for use with ***for_each***

Why Do We Need These Functions?

Imagine managing **multiple environments**:

Requirement	**Example**
Consistent Naming	`web-01`, `web-02`
Dynamic Config Files	Scripts change by environment
Combined Tags	Base + environment + resource
Flexible Security Rules	Multiple ports per rule group

The Limitation of Basic Interpolation

Basic interpolation works for simple strings:

```
"${var.project}-web"
```

But fails when we need:

❌ Zero-padded numbering (web-01)
❌ Loops in configuration files
❌ Combining multiple maps of tags
❌ Converting nested structures into flat lists

The Solution: Terraform Built-In Functions

Terraform includes powerful data transformation functions:

Function	**Purpose**
`format()`	Advanced string formatting
`templatefile()`	Render dynamic files
`zipmap()`	Build maps from lists
`merge()`	Combine maps
`flatten()`	Flatten nested lists

Quick Note: *path.module*

path.module gives the directory path of the current module.

```
templatefile("${path.module}/templates/config.tftpl", {...})
```

Format()

When You'll Use These in Real Projects

Scenario	Function Used
Naming EC2 instances	`format()`
Creating user data scripts	`templatefile()`
Mapping tiers to instance types	`zipmap()`
Combining tag layers	`merge()`
Breaking nested rules into individual rules	`flatten()`

Format()

Why Naming Standards Matter

Imagine a production environment:

Without Standard	With Standard
web1	web-01
web2	web-02
db1	db-01

Problems Without Formatting

❌ Hard to sort resources
❌ Hard to automate monitoring
❌ Hard to enforce conventions

Interpolation vs *format()*

Use Interpolation for Simple Joining

```
"${var.project}-${var.environment}-web"
```

Result:

```
myapp-dev-web
```

Use `format()` When we Need Special Formatting

```
format("%s-web-%02d", var.project, 1)
```

Result:

```
myapp-web-01
```

Understanding format() Syntax

```
format(format_string, value1, value2, ...)
```

The **format string** contains placeholders that get replaced with values.

Placeholder	**Meaning**
`%s`	String
`%d`	Decimal number
`%02d`	Zero-padded number (2 digits)

Format()

Breaking Down %02d

%02d means:

Symbol	Meaning
`%`	Start placeholder
`0`	Pad with zeros
`2`	Minimum width 2 digits
`d`	Decimal number

Examples

Code	Output
`format("server-%02d", 1)`	server-01
`format("server-%02d", 9)`	server-09
`format("server-%02d", 12)`	server-12 (already 2 digits, no padding needed)
`format("server-%02d", 100)`	server-100 (already 3 digits, no padding needed)

Format()

Real-World Naming Pattern

```
locals {
	name_prefix = "lab3-${var.project}-${var.environment}"
	formatted_server_name = format("%s-web-%02d", local.name_prefix, 1)
}
```

Result:

```
lab3-user1-dev-web-01
```

Format()

Breaking Down %02d

%02d means:

Symbol	Meaning
`%`	Start placeholder
`0`	Pad with zeros
`2`	Minimum width 2 digits
`d`	Decimal number

Examples

Code	Output
`format("server-%02d", 1)`	server-01
`format("server-%02d", 9)`	server-09
`format("server-%02d", 12)`	server-12 (already 2 digits, no padding needed)
`format("server-%02d", 100)`	server-100 (already 3 digits, no padding needed)

Why Do We Need Templates?

Imagine launching EC2 instances across environments:

Environment	Config Difference
Dev	Debug tools enabled
Staging	Monitoring enabled
Prod	Hardened security

Hardcoding scripts per environment = ❌ Not scalable

What is templatefile()?

templatefile() reads a file and replaces placeholders with Terraform values.

```
templatefile(path, variables_map)
```

Parameter	Meaning
path	Path to template file
variables_map	Values to inject

Template File Basics

Template file example **(user_data.tftpl):**

```
#!/bin/bash
ENVIRONMENT="${environment}"
SERVER_NAME="${server_name}"
echo "Setting up $SERVER_NAME in $ENVIRONMENT"
```

Syntax	Purpose
`${variable}`	Insert value

Templates

Template File Basics

Template file example **(user_data.tftpl):**

```
#!/bin/bash
ENVIRONMENT="${environment}"
SERVER_NAME="${server_name}"
echo "Setting up $SERVER_NAME in $ENVIRONMENT"
```

Syntax	Purpose
`${variable}`	Insert value

Rendering the Template in Terraform

```
locals {
  user_data = templatefile("${path.module}/templates/user_data.tftpl", {
    environment = var.environment
    server_name = "web-server"
  })
}
```

Result (environment = dev)

```
ENVIRONMENT="dev"
SERVER_NAME="web-server"
```

Template Loops

Templates support loops for repeating sections.

Template:

```
#!/bin/bash
# Enable services
%{ for service in services ~}
systemctl enable ${service}
systemctl start ${service}
%{ endfor ~}
```

Output:

```
systemctl enable httpd
systemctl start httpd
systemctl enable sshd
systemctl start sshd
```

Terraform:

```
services = ["httpd", "sshd"]
```

Understanding ~} in Templates

- **The tilde** (~}) removes extra newlines after loop blocks.
- **Without** ~} → extra blank lines
- **With** ~} → clean formatting

Using *templatefile()* in Resources

```
resource "aws_instance" "web" {
    ami           = data.aws_ami.amazon_linux.id
    instance_type = "t3.nano"
    user_data     = local.user_data
}
```

zipmap()

The Problem: Parallel Lists

Imagine we define server tiers and their instance sizes separately:

```
tiers        = ["web", "db", "cache"]
server_types = ["t3.nano", "t3.micro", "t3.small"]
```

These lists are related by **position**, but Terraform cannot automatically link them.

Challenges:

❌ Hard to read
❌ Easy to mismatch values
❌ Not scalable

zipmap()

Creating Maps from Lists with zipmap()

The **zipmap()** function creates a map from two parallel lists - one list of keys and one list of values.

The Format

```
zipmap(keys_list, values_list)
```

Both lists must have the same length. The first item in the keys list pairs with the first item in the values list, and so on.

Basic Example

```
zipmap(["a", "b", "c"], [1, 2, 3])
# Result: { a = 1, b = 2, c = 3 }
```

zipmap()

Step-by-Step Mapping

Position	Tier	Instance Type
1	web	t3.nano
2	db	t3.micro
3	cache	t3.small

Terraform Code

```
zipmap(var.tiers, var.server_types)
```

Output Example

```
{
  web   = "t3.nano"
  db    = "t3.micro"
  cache = "t3.small"
}
```

zipmap()

Practical Example: Instance Type Lookup

Imagine we have two related lists - server tiers and their corresponding instance types:

```
variable "tiers" {
	default = ["web", "db", "cache"]
}
variable "server_types" {
	default = ["t3.nano", "t3.micro", "t3.small"]
}
```

Use zipmap() to create a lookup table:

```
locals {
	tier_instance_types = zipmap(var.tiers,
	var.server_types)

	# Result: { web = "t3.nano", db = "t3.micro",
	cache = "t3.small" }
}
```

Now we can look up instance types by tier name:

```
resource "aws_instance" "web" {
	ami = data.aws_ami.amazon_linux.id
	instance_type =
	local.tier_instance_types["web"] # "t3.nano"
}
```

zipmap()

When to Use zipmap()

Use ***zipmap()*** when we have:

- we have **two related lists**
- Data is **positionally aligned**
- we need **name-based lookups**
- we want cleaner resource configuration

zipmap()

Important Rule: Lists Must Match

The lists must be the **same length**.

```
zipmap(["a","b"], [1,2,3]) ❌ Error
```

Why?

Terraform wouldn't know what value to assign to extra keys.

zipmap()

Real-World Use Cases

Scenario	Example
Tier to instance mapping	web → t3.nano
Region to AMI ID mapping	us-east-1 → ami-abc
Environment to CIDR mapping	dev → 10.0.1.0/24

merge()

The Real-World Problem

Imagine tagging strategy for cloud resources:

Layer	Example
Organization-wide tags	`ManagedBy=Terraform`
Environment tags	`Environment=Dev`
Resource-specific tags	`Name=web-01`

Without `merge()`:

❌ Tags scattered
❌ Duplicate definitions
❌ Hard to maintain consistency

merge()

Important Rule: Lists Must Match

merge() combines multiple maps into one.

```
merge(map1, map2, map3, ...)
```

If the same key appears multiple times → **last map wins**

merge()

Basic Example

```
merge(
  { a = 1, b = 2 },
  { b = 3, c = 4 }
)
```

Result:

```
{ a = 1, b = 3, c = 4 }
```

merge()

Practical Example: Building Tags

```
variable "common_tags" {
  default = {
    ManagedBy = "Terraform"
    Team      = "Platform"
  }
}

variable "environment_tags" {
  default = {
    CostCenter = "development"
    Compliance = "standard"
  }
}
```

merge()

Layering Tags with merge()

```
locals {
  all_tags = merge(
    var.common_tags,
    var.environment_tags,
    {
      Name        = local.server_name
      Environment = var.environment
    }
  )
}
```

Resulting Tags

ManagedBy = Terraform
Team = Platform
CostCenter = development
Compliance = standard
Name = lab3-user1-dev-web
Environment = dev

merge()

Override Behavior (Very Important)

```
merge(
  { Environment = "default" },
  { Environment = "dev" }
)
```

Result:

```
Environment = dev
```

Rule

Later maps override earlier ones

merge()

Default + Override Pattern

```
locals {
  default_tags = {
    Environment = "unknown"
    ManagedBy   = "Terraform"
  }

  final_tags = merge(
    local.default_tags,
    var.user_provided_tags
  )
}
```

Using merge() in Resources

```
resource "aws_instance" "web" {
  instance_type = "t3.nano"

  tags = merge(local.all_tags, {
    Name = "${local.server_name}-web"
  })
}
```

merge()

Real-World Use Cases

Scenario	Example
Tagging strategy	Org + Env + Resource tags
IAM policies	Base policy + team overrides
Security rules	Default rules + app-specific rules
Metadata	Global labels + custom labels

flatten()

The Problem: Nested Data

```
security_group_rules = [
 {
   name  = "web"
   ports = [80, 443]
 },
 {
   name  = "db"
   ports = [3306, 5432]
 }
]
```

Easy for Humans

✓ Groups related ports

Hard for Terraform

❌ `for_each` cannot iterate nested lists directly

flatten()

What Does flatten() Do?

```
flatten([[1,2],[3,4]])
```

Result:

```
[1,2,3,4]
```

flatten() Transforming Rule Groups into Individual Rules

We want to convert:

```
web → [80,443]
db  → [3306,5432]
```

Into:

```
web-80
web-443
db-3306
db-5432
```

flatten()

Nested for Expression Pattern

```
locals {
  flattened_rules = flatten([
    for rule in var.security_group_rules : [
      for port in rule.ports : {
        key  = "${rule.name}-${port}"
        name = rule.name
        port = port
      }
    ]
  ])
}
```

flatten()

Resulting Flat List

```
[
  { key = "web-80", name = "web", port = 80 },
  { key = "web-443", name = "web", port = 443 },
  { key = "db-3306", name = "db", port = 3306 },
  { key = "db-5432", name = "db", port = 5432 }
]
```

flatten()

Converting List to Map for for_each

`for_each` requires a **map or set**, not a list.

```
locals {
  ingress_rule_map = {
    for rule in local.flattened_rules :
    rule.key => rule
  }
}
```

Result:

```
{
  "web-80"  = { name="web", port=80 }
  "web-443" = { name="web", port=443 }
  "db-3306" = { name="db", port=3306 }
}
```

flatten()

Using Flattened Data in a Resource

```
resource "aws_vpc_security_group_ingress_rule" "rules" {
    for_each = local.ingress_rule_map

    description     = each.value.name
    from_port       = each.value.port
    to_port         = each.value.port
    ip_protocol     = "tcp"
    cidr_ipv4       = "0.0.0.0/0"
}
```

Key Takeaways

- **Use interpolation for simple string joining** ("${var.a}-${var.b}"), but use **format()** when we need special formatting like zero-padding
- **templatefile()** renders files with variable substitution and loops - perfect for user data scripts, configuration files, and any dynamic text content
- **zipmap()** creates a map from two parallel lists - useful for creating lookup tables from related data
- **merge()** combines maps with later values overriding earlier ones - the standard pattern for layering tags from multiple sources
- **Order matters in merge()** - put your defaults first and overrides last
- **flatten()** converts nested lists into flat lists - essential when your variable structure is hierarchical but we need individual items for for_each

Knowledge Check

Question 1

- What is the result of this expression?

```
format("server-%02d", 5)
```

A)"server-5"
B)"server-05"
C)"server-005"
D)"server-%02d5"

Solution:
B - The **%02d** format specifier means "decimal number, zero-padded to 2 digits". The number **5 becomes "05",** so the full result is "server-05".

Knowledge Check

Question 2

- Given this nested structure:

```
rules = [
    { name = "http", ports = [80] },
    { name = "https", ports = [443] }
]
```

Options:

A) 1

B) 2

C) 3

D) 4

After using the flatten pattern, how many items are in the resulting flat list?

Solution:

B - Each rule has one port, so flattening produces 2 items total: one for http port 80 and one for https port 443. The flatten pattern creates one item per port, not per rule group.

Knowledge Check

Question 3

- What is the result of this merge operation?

```
merge(
{ Environment = "dev", Team = "alpha" },
{ Environment = "staging", Owner = "bob" }
)
```

A) { Environment = "dev", Team = "alpha", Owner = "bob" }
B) { Environment = "staging", Team = "alpha", Owner = "bob" }
C) An error because Environment appears twice
D) { Environment = ["dev", "staging"], Team = "alpha", Owner = "bob" }

Solution:

B - The merge() function combines maps, and when the same key appears multiple times, the last value wins. Environment appears in both maps, so the second map's value ("staging") overrides the first map's value ("dev"). The result includes all unique keys with the last-seen values.