Module 3: Advanced Terraform Functions

Learning Objectives

By the end of this module you will understand:

- How to use string formatting functions to create consistent naming patterns
- How to render dynamic configuration files from templates
- How to create and combine maps for flexible configuration management
- How to transform nested data structures into flat lists for use with `for_each`

Why Do We Need These Functions?

Imagine you're managing infrastructure across multiple environments. Each environment needs:

- **Consistent naming** - Servers named "web-01", "web-02" with zero-padded numbers
- **Dynamic configuration files** - User data scripts that change based on environment
- **Combined tags** - Base tags merged with environment-specific tags and resource-specific tags
- **Flexible security rules** - Nested rule definitions that need to be flattened for individual ingress rules

The Problem: Basic string interpolation (`"${var.name}-web"`) works for simple cases, but falls short when you need zero-padding, template loops, or need to transform complex data structures.

The Solution: Terraform provides built-in functions that transform, format, and combine data:

- `format()` - String formatting with placeholders
- `templatefile()` - Render files with embedded variables and loops
- `zipmap()` - Create a map from two parallel lists
- `merge()` - Combine multiple maps into one
- `flatten()` - Convert nested lists into flat lists

Quick Note: path.module

You'll see `path.module` in the examples below. This is a built-in Terraform value that gives you the path to the directory where the current module's configuration files are located.

```
# If your .tf files are in /home/user/lab3/
templatefile("${path.module}/templates/config.tftpl", { ... })
# Resolves to: /home/user/lab3/templates/config.tftpl
```

Using `path.module` ensures your file paths work correctly regardless of where you run Terraform from.

String Formatting with format()

The `format()` function lets you create strings using special placeholders that control how values are inserted.

When to Use format() vs Interpolation

For simple string joining, use regular interpolation:

```
# Simple joining - use interpolation
server_name = "${var.project}-${var.environment}-web"
# Result: "myapp-dev-web"
```

Use `format()` when you need special formatting like zero-padding:

```
# Zero-padding - use format()
server_name = format("%s-web-%02d", var.project, 1)
# Result: "myapp-web-01"
```

The Format

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

The format string contains placeholders that get replaced with values:

- `%s` - Insert a string value
- `%d` - Insert a number
- `%02d` - Insert a number with zero-padding to 2 digits

How %02d Works

The `%02d` placeholder means:

- `%` - Start of a placeholder
- `0` - Pad with zeros (instead of spaces)
- `2` - Make the result at least 2 characters wide
- `d` - The value is a decimal number

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

Practical Example

```
locals {
  name_prefix = "lab3-${var.project}-${var.environment}"

  # Simple joining - interpolation is fine
  server_name = "${local.name_prefix}-web"
  # Result: "lab3-user1-dev-web"

  # Zero-padded numbering - use format()
  formatted_server_name = format("%s-web-%02d", local.name_prefix, 1)
  # Result: "lab3-user1-dev-web-01"
}
```

This says: "Take the name prefix, add '-web-', then add the number 1 padded to 2 digits."

Dynamic File Generation with templatefile()

The `templatefile()` function reads a file and replaces placeholders with values you provide. This is powerful for generating configuration files, scripts, or any text that needs to change based on your Terraform variables.

The Format

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

- `path` - The file to read (usually with `.tftpl` extension)
- `variables_map` - A map of variable names and their values

Template File Syntax

Template files use special syntax for placeholders:

- `${variable}` - Insert a variable value
- `%{ for item in list ~}` ... `%{ endfor ~}` - Loop over a list

Basic Example

Template file (`templates/user_data.tftpl`):

```
#!/bin/bash
ENVIRONMENT="${environment}"
SERVER_NAME="${server_name}"

echo "Setting up $SERVER_NAME in $ENVIRONMENT"
```

Terraform code:

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

Result (when environment = "dev"):

```
#!/bin/bash
ENVIRONMENT="dev"
SERVER_NAME="web-server"

echo "Setting up web-server in dev"
```

Template Loops

Templates can loop over lists using the `%{ for ... }` directive:

Template file:

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

Terraform code:

```
locals {
  user_data = templatefile("${path.module}/templates/user_data.tftpl", {
    services = ["httpd", "sshd"]
  })
}
```

Result:

```
#!/bin/bash
# Enable services
systemctl enable httpd
systemctl start httpd
systemctl enable sshd
systemctl start sshd
```

The `~}` at the end of template directives removes the newline after the directive, keeping the output clean.

Using templatefile() in Resources

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

  # Pass the rendered template as user_data
  user_data = local.user_data
}
```

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 }
```

Practical Example: Instance Type Lookup

Imagine you 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 you 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"
}
```

When to Use zipmap()

Use `zipmap()` when you have:

- Two lists that correspond to each other positionally
- A need to look up values by name rather than by position
- Data from external sources that comes as parallel lists

Combining Maps with merge()

The `merge()` function combines multiple maps into a single map. If the same key appears in multiple maps, the value from the **last** map wins.

The Format

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

Basic Example

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

Notice that key `b` appears in both maps. The second map's value (`3`) overrides the first map's value (`2`).

Practical Example: Building Tags

A common pattern is layering tags from different sources:

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

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

locals {
  all_tags = merge(
    var.common_tags,                    # Base tags
    var.environment_tags,               # Environment-specific tags
    {
      Name        = local.server_name  # Resource-specific tags
      Environment = var.environment
    }
  )
}
```

Result:

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

Override Behavior

The order matters. Later maps override earlier ones:

```
merge(
  { Environment = "default" },  # Base value
  { Environment = "dev" }       # Override
)
# Result: { Environment = "dev" }
```

This is useful for providing defaults that can be overridden:

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

  final_tags = merge(
    local.default_tags,
    var.user_provided_tags  # User can override defaults
  )
}
```

Using merge() in Resources

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

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

This takes all the common tags and adds a Name tag specific to this resource.

Flattening Nested Data with flatten()

The `flatten()` function converts nested lists (lists inside lists) into a single flat list. This is essential when you have hierarchical data but need individual items for `for_each`.

The Problem

Imagine you define security rules where each rule group has multiple ports:

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

This nested structure is easy to read, but `for_each` needs individual rules - one per port.

Basic flatten() Behavior

```
flatten([[1, 2], [3, 4]])
# Result: [1, 2, 3, 4]

flatten([["a", "b"], ["c"]])
# Result: ["a", "b", "c"]
```

The Common Pattern

To convert nested rule groups into individual rules, use a nested for expression with flatten:

```
locals {
  # Input: [{name="web", ports=[80,443]}, {name="db", ports=[3306]}]
  # Output: [{name="web", port=80}, {name="web", port=443}, {name="db",
port=3306}]

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

Let's break this down step by step:

1. The outer `for rule in var.security_group_rules` loops through each rule group
2. The inner `for port in rule.ports` loops through each port in that group
3. For each combination, we create an object with the rule name and individual port
4. The `key` field creates a unique identifier like "web-80" or "db-3306"
5. `flatten()` takes the nested result and makes it a single flat list

Result with the example input:

```
[
  { 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 }
]
```

Converting to a Map for for_each

The `for_each` meta-argument requires a map or set, not a list. Convert the flattened list to a map:

```
locals {
  # Convert list to map using the key field
  ingress_rule_map = {
    for rule in local.flattened_rules : rule.key => rule
  }
}
```

Result:

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

Using the Flattened Data

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

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

This creates four separate ingress rules - one for each port - from the original nested definition.

Key Takeaways

1. **Use interpolation for simple string joining** (`"${var.a}-${var.b}"`), but use **format()** when you need special formatting like zero-padding
2. **templatefile()** renders files with variable substitution and loops - perfect for user data scripts, configuration files, and any dynamic text content
3. **zipmap()** creates a map from two parallel lists - useful for creating lookup tables from related data

4. **merge()** combines maps with later values overriding earlier ones - the standard pattern for layering tags from multiple sources
5. **Order matters in merge()** - put your defaults first and overrides last
6. **flatten()** converts nested lists into flat lists - essential when your variable structure is hierarchical but you 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"`

Question 2

Given this nested structure:

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

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

A) 1 B) 2 C) 3 D) 4

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" }`

Answers

Question 1: 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".

Question 2: 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.

Question 3: 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.

Dictionary of Terms

flatten() - A function that converts nested lists (lists containing lists) into a single flat list. Example: `flatten([[1,2], [3,4]])` produces `[1, 2, 3, 4]`.

format() - A function that creates strings using format specifiers (placeholders) that control how values are inserted. Supports features like zero-padding that simple interpolation cannot do.

Format Specifier - A placeholder in a format string that controls how a value is inserted. Common specifiers: `%s` (string), `%d` (decimal number), `%02d` (zero-padded to 2 digits).

merge() - A function that combines multiple maps into one. When keys conflict, values from later maps override earlier ones.

path.module - A built-in Terraform value that contains the filesystem path to the directory where the current module's configuration files are located.

Template Directive - Special syntax in template files that controls flow, such as `%{ for item in list ~}` for loops. Directives are processed when the template is rendered.

templatefile() - A function that reads a template file and returns the rendered result after replacing placeholders with provided values.

tftpl - The recommended file extension for Terraform template files. Using `.tftpl` helps editors provide syntax highlighting and makes the file's purpose clear.

zipmap() - A function that creates a map from two lists of equal length. The first list provides the keys, the second list provides the values.

Documentation Links

- format Function
- templatefile Function
- zipmap Function
- merge Function
- flatten Function