Module 2: Dynamic Infrastructure with for_each

Terraform-Intermediate-Day1-Module2

Learning Objectives

By the end of this module you will understand:

- How to use for_each to create multiple resources from a single resource block
- How to work with maps of objects for complex configuration data
- How for_each handles additions and removals gracefully compared to count
- How to reference resources created with for_each

Why Do We Need for_each?

Imagine managing multiple servers:

- web-1
- web-2
- db-1

```
resource "aws_instance" "web_1" {
      ami = data.aws_ami.amazon_linux.id
      instance_type = "t3.nano"
      tags = { Name = "web-1", Tier = "web" }
}
resource "aws_instance" "web_2" {
      ami = data.aws_ami.amazon_linux.id
      instance_type = "t3.nano"
      tags = { Name = "web-2", Tier = "web" }
}
resource "aws_instance" "db_1" {
      ami = data.aws_ami.amazon_linux.id
      instance_type = "t3.micro"
      tags = { Name = "db-1", Tier = "db" }
}
```

Why Do We Need for_each?

The Problem: This approach has several issues:

Without `for_each`, you write separate blocks
for each server.

Problem:

❌ Copy-paste code
❌ Hard to add/remove servers
❌ Easy to make mistakes
❌ Config spread across many places

Could We Use count?

We learned **count** in Module 1. Could it help here?

```
variable "server_count" {
    default = 3
}
resource "aws_instance" "servers" {
    count = var.server_count
    ami = data.aws_ami.amazon_linux.id
    instance_type = "t3.nano"
    tags = {
        Name = "server-${count.index}"
    }
}
```

Could We Use count? ….

This reduces repetition, but count tracks them as :

```
servers[0], servers[1], servers[2]
```

The Problems:

- All instances must be similar
- Identified by index numbers
- Removing one shifts indexes
- Can cause unexpected recreation

The Solution: *for_each*

for_each lets us define infrastructure as **data with names**.

✓ Each server has a unique key
✓ Each server can have different configuration
✓ Add or remove safely

Quick Note: Sets and toset()

- Before we dive into for_each, we need to understand sets.
- A **set** is a collection where every item is unique and items have no particular order. This is different from a **list**, where **items** can **repeat** and **order matters**.
- for_each requires either a map or a set - it cannot work with a plain list.
- The toset() function converts a list into a set:

```
# A list (can have duplicates, order matters)
    server_tiers_list = ["web", "db", "web"]

# Convert to set (removes duplicates, no order)
    server_tiers_set = toset(["web", "db", "web"])

# Result: {"web", "db"} - the duplicate "web" is gone
```

Quick Note: Maps vs Objects

What is an Object?

- An object is like a form with specific fields to fill out. Each field has a name and expects a certain type of information.

```
# An object type definition - like a blank form
type = object({
        instance_type = string # Field 1: expects text
        tier = string # Field 2: expects text
        port = number # Field 3: expects a number
})
```

```
# A filled-out form
{
        instance_type = "t3.nano"
        tier = "web"
        port = 80
}
```

The key point: **object fields are fixed**. You define them in advance, and every object of that type has the same fields.

Quick Note: Maps vs Objects

What is an Map?

- A **map** is like a contact list or phone book. You can add any name (key) you want, and each name points to some information (value).

```
# A map of strings - like a phone book
	type = map(string)

# Example: any names, all values are phone numbers (strings)
{
	"Alice" = "555-1234"
	"Bob" = "555-5678"
	"Carol" = "555-9999"
}
```

The key point: **map keys are flexible**. You can add "Alice", "Bob", or any other name - they're not defined in advance. But all values must be the same type.

Quick Note: Maps vs Objects

What is a Map of Objects?

- A map of objects combines both: **flexible keys** (like a contact list) where each entry contains **structured data** (like a form).
- This is powerful because you can add as many servers as you want (flexible keys), and each server has consistent, structured configuration (fixed object fields).

```
# Type: a map where each value is an object

Variable "servers" {
    type = map(object({
        instance_type = string
        tier = string
    }))
}
```

Quick Note: Maps vs Objects

```
# Example: contact list where each person has a filled-out form

servers = {
    "web-1" = {             # Key: "web-1" (you choose the name)
        instance_type = "t3.nano" # Object field 1
        tier = "web" # Object field 2
    }

    "db-1" = {              # Key: "db-1" (you choose the name)
        instance_type = "t3.micro"
        tier = "db"
    }
}
```

The for_each Meta-Argument

for_each tells Terraform to create one copy of a resource for each item in a map or set.

The Format

```
resource "resource_type" "name" {
        for_each = <map or set>
        # Use each.key and each.value to access the current item
        some_attribute = each.key
        other_attribute = each.value
}
```

What Are each.key and each.value?

When Terraform loops through the data, it gives two special values for the current item:

- each.key - The identifier for the current item (the map key or set value)
- each.value - The data associated with that key (for maps) or the same as the key (for sets)

Using for_each with a Set

The simplest use of for_each is with a set of strings.

Example: Create a security group for each unique tier:

```
locals {
    # Get unique tiers from our servers
    server_tiers = toset(["web", "db"])
}
resource "aws_security_group" "tier_sg" {
    for_each = local.server_tiers
    # each.key is the tier name ("web" or "db")
    name = "${local.name_prefix}-${each.key}-sg"
    description = "Security group for ${each.key} tier"
}
```

The for_each Meta-Argument

How it works step by step:

- Terraform sees for_each = local.server_tiers which contains {"web", "db"}
- For the first iteration: each.key = "web", each.value = "web" (same for sets)
- Terraform creates aws_security_group.tier_sg["web"]
- For the second iteration: each.key = "db", each.value = "db"
- Terraform creates aws_security_group.tier_sg["db"]

Result: Two security groups are created, one for each tier.

Using for_each with a Map of Objects

Extracting Unique Values with for Expressions

Goal: One security group per tier, not per server.

```
locals {
  server_tiers = toset([for s in var.servers : s.tier])
}
```

Result:

```
{"web", "db"}
```

Referencing for_each Resources

```
aws_security_group.tier_sg["web"].id
```

Dynamic example:

```
aws_security_group.tier_sg[each.value.tier].id
```

for_each vs count: When to Use Each

Feature	count	for_each
Identifier	Index	Key
Safe removal	❌	✅
Unique config	Hard	Easy
Stable references	❌	✅

Combining count with contains()

```
count = contains(local.server_tiers, "web") ? 1 : 0
```

Outputs with for_each

```
output "server_details" {
  value = {
    for name, instance in aws_instance.servers :
    name => instance.private_ip
  }
}
```

Outputs with for_each

```
output "server_details" {
  value = {
    for name, instance in aws_instance.servers :
    name => {
      id         = instance.id
      private_ip = instance.private_ip
      tier       = instance.tags["Tier"]
    }
  }
}
```

Key Takeaways

1. **for_each** creates one copy of a resource for each item in a map or set, letting you define infrastructure as data
2. **each.key** gives you the current item's identifier; **each.value** gives you its data
3. **Maps of objects** let you store complex, multi-attribute configuration for each resource
4. **toset()** converts a list to a set, removing duplicates - useful for extracting unique values
5. **for_each resources are referenced by key** *(resource["key"])*, not by index like count *(resource[0])*
6. **for_each handles changes gracefully** - adding or removing items only affects those specific resources, not others in the collection

Knowledge Check

Question 1

- Given this configuration:

```
variable "servers" {
	type = map(object({
		instance_type = string
		tier = string
	}))
}
servers = {
	"api-1" = { instance_type = "t3.small", tier = "api" }
	"web-1" = { instance_type = "t3.nano", tier = "web" }
}
```

What is the value of each.value.tier when processing the "api-1" server?

A) "api-1"

B) "api"

C) { instance_type = "t3.small", tier = "api" }

D) "t3.small"

Solution:

B - each.key is "api-1" (the map key), and each.value is the entire object { instance_type = "t3.small", tier = "api" }. So each.value.tier accesses the tier attribute, which is "api".

Knowledge Check

Question 2

- What happens when you remove a server from a for_each map compared to removing an item from a count list?

A) Both behave the same way - all remaining resources are recreated
B) for_each only destroys the removed resource; count may renumber and recreate others
C) count only destroys the removed resource; for_each may recreate others
D) Neither approach allows removing resources

Solution:

B - for_each resources are identified by their keys, so removing "web-2" only affects that specific resource. With count, resources are identified by index numbers, so removing an item can cause renumbering that affects other resources.

Knowledge Check

Question 3

- Why do we use toset() when extracting unique tiers from a servers map?

```
server_tiers = toset([for server in var.servers : server.tier])
```

A) Because for_each requires a set, not a list
B) To sort the tiers alphabetically
C) To remove duplicate tier values since multiple servers can share the same tier
D) Both A and C are correct

Solution:

D - Both reasons are correct. The for expression ***[for server in var.servers : server.tier]*** produces a list that might have duplicates ***(e.g., ["web", "web", "db"]). toset()*** removes duplicates AND converts to a set, which is required because ***for_each*** cannot iterate over a plain list.