Module 2: Dynamic Infrastructure with for_each

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 you're managing a web application with multiple servers. Each server has a name, instance type, and belongs to a tier (web or db).

Without for_each, you would write separate resource blocks for each server:

```
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" }
}
```

The Problem: This approach has several issues:

- Adding a new server means copying and pasting an entire resource block
- Removing a server means finding and deleting the right block
- Configuration is scattered across multiple blocks instead of being in one place
- Easy to make mistakes when servers have similar but slightly different settings

Could We Use count?

You 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}" }
}
```

This reduces repetition, but `count` has limitations:

- **All instances must be nearly identical** - you can't easily give server-0 a different instance type than server-2
- **Instances are identified by number** - they become `servers[0]`, `servers[1]`, `servers[2]`
- **Removing an item causes problems** - if you delete the middle server, all indexes after it shift down

Example of the index problem: You have servers [0], [1], [2]. You remove server [1]. Now server [2] becomes server [1]. Terraform sees this as: "destroy old [1], modify [2] to become new [1], destroy [2]". Two servers are affected when you only wanted to remove one.

The Solution: `for_each` solves these problems. It lets you define servers as data with unique names, and each server can have different configuration. Add a server? Just add a line to your data. Remove one? Only that specific server is destroyed - others are unaffected.

Quick Note: Sets and toset()

Before we dive into `for_each`, you 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
```

You'll see `toset()` used when you need to iterate over unique values extracted from a collection.

Quick Note: Maps vs Objects

Before working with `for_each`, it helps to understand the difference between maps and objects in Terraform.

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.

What is a 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.

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).

```
# Type: a map where each value is an object
type = map(object({
  instance_type = string
  tier          = string
}))

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

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).

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 your data, it gives you 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"
}
```

How it works step by step:

1. Terraform sees `for_each = local.server_tiers` which contains `{"web", "db"}`
2. For the first iteration: `each.key = "web"`, `each.value = "web"` (same for sets)
3. Terraform creates `aws_security_group.tier_sg["web"]`
4. For the second iteration: `each.key = "db"`, `each.value = "db"`
5. 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

Maps of objects let you store complex configuration data for each item.

Defining the Variable

```
variable "servers" {
  description = "Map of server configurations"
  type = map(object({
    instance_type = string
    tier          = string
  }))
}
```

This says: "servers is a map where each key is a server name, and each value is an object containing instance_type and tier."

Providing the Data

In your tfvars file:

```
servers = {
  "web-1" = {
```

```
    instance_type = "t3.nano"
    tier          = "web"
  }
  "web-2" = {
    instance_type = "t3.nano"
    tier          = "web"
  }
  "db-1" = {
    instance_type = "t3.micro"
    tier          = "db"
  }
}
```

Creating Resources from the Map

```
resource "aws_instance" "servers" {
  for_each = var.servers

  ami           = data.aws_ami.amazon_linux.id
  instance_type = each.value.instance_type  # Access the instance_type
from the object

  tags = {
    Name = "${local.name_prefix}-${each.key}"  # each.key is the server
name
    Tier = each.value.tier                      # Access the tier from the
object
  }
}
```

How it works step by step:

For the "web-1" iteration:

- `each.key` = `"web-1"` (the map key)
- `each.value` = `{ instance_type = "t3.nano", tier = "web" }` (the entire object)
- `each.value.instance_type` = `"t3.nano"`
- `each.value.tier` = `"web"`

Terraform creates `aws_instance.servers["web-1"]` with these values.

Extracting Unique Values with for Expressions

Often you need to extract unique values from your data. For example, creating one security group per tier, not per server.

```
locals {
  # Extract just the tier values and convert to a set
```

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

Let's break down the for expression `[for server in var.servers : server.tier]`:

- `[` and `]` - Square brackets mean "produce a list"
- `for server in var.servers` - Loop through each item in `var.servers`, calling each item `server` (you choose this temporary name)
- `:` - Separates the loop definition from the output expression
- `server.tier` - What to output for each item (in this case, just the tier value)

Then `toset(...)` wraps the whole thing to convert that list to a set, removing duplicates.

Given this input:

```
servers = {
  "web-1" = { tier = "web", ... }
  "web-2" = { tier = "web", ... }
  "db-1"  = { tier = "db", ... }
}
```

The result is:

```
server_tiers = {"web", "db"}  # Set with unique tiers only
```

Referencing for_each Resources

When you use `for_each`, Terraform creates a map of resources, not a list. You reference them using the key in square brackets.

Single Resource Reference

```
# Reference a specific security group by tier name
security_group_id = aws_security_group.tier_sg["web"].id
```

Dynamic Reference

When creating instances, you need to assign each server to its tier's security group:

```
resource "aws_instance" "servers" {
  for_each = var.servers

  ami           = data.aws_ami.amazon_linux.id
```

```
  instance_type = each.value.instance_type

  # Look up the security group for this server's tier
  vpc_security_group_ids =
[aws_security_group.tier_sg[each.value.tier].id]
}
```

This says: "For the current server, get its tier from `each.value.tier`, then look up the security group with that key."

---

for_each vs count: When to Use Each

Both `for_each` and `count` create multiple resources, but they behave differently.

count Creates a List (Indexed by Numbers)

```
resource "aws_instance" "servers" {
  count = 3
  # Creates: aws_instance.servers[0], aws_instance.servers[1],
aws_instance.servers[2]
}
```

Problem with count: If you remove the middle item, everything after it gets renumbered. Removing item [1] causes item [2] to become the new [1], which Terraform sees as a change to [1] plus deletion of [2].

for_each Creates a Map (Indexed by Keys)

```
resource "aws_instance" "servers" {
  for_each = var.servers
  # Creates: aws_instance.servers["web-1"], aws_instance.servers["web-2"],
aws_instance.servers["db-1"]
}
```

Advantage of for_each: If you remove "web-2", only that specific resource is destroyed. "web-1" and "db-1" are completely unaffected because their keys haven't changed.

When to Use Which

Use **count** when:

- You need a simple on/off switch (`count = var.enabled ? 1 : 0`)
- Resources are truly identical and interchangeable
- You're creating a fixed number of identical resources

Use **for_each** when:

- Each resource has unique configuration
- Resources need stable identifiers
- You expect to add or remove items over time
- Resources are referenced by other parts of your configuration

---

Combining count with contains()

Sometimes you need to conditionally create a resource based on whether a value exists in a collection. The `contains()` function is perfect for this.

The contains() Function

`contains()` checks if a list or set includes a specific value and returns `true` or `false`:

```
contains(["web", "db"], "web")  # Returns: true
contains(["web", "db"], "api")  # Returns: false
```

The Pattern: count with contains()

Combine `contains()` with `count` to create a resource only when a condition is met:

```
resource "aws_vpc_security_group_ingress_rule" "web_http" {
  # Create this rule only if "web" tier exists
  count = contains(local.server_tiers, "web") ? 1 : 0

  security_group_id = aws_security_group.tier_sg["web"].id
  from_port         = 80
  to_port           = 80
  ip_protocol       = "tcp"
  cidr_ipv4         = "0.0.0.0/0"
}
```

How it works:

1. `contains(local.server_tiers, "web")` checks if "web" is in the set
2. If true, the ternary returns `1`, creating one resource
3. If false, the ternary returns `0`, creating no resource

This pattern is useful when different items need different configurations that can't be generalized with `for_each`.

Outputs with for_each

When outputting information about `for_each` resources, use a for expression to format the data:

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

This creates a map where each key is the server name and each value is an object with the details.

Example output:

```
server_details = {
  "db-1" = {
    id            = "i-0abc123..."
    instance_type = "t3.micro"
    private_ip    = "172.31.1.10"
    tier          = "db"
  }
  "web-1" = {
    id            = "i-0def456..."
    instance_type = "t3.nano"
    private_ip    = "172.31.1.11"
    tier          = "web"
  }
}
```

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"`

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

---

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

---

Answers

Question 1: 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"`.

Question 2: 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.

Question 3: 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.

Dictionary of Terms

each.key - A special value available inside a `for_each` block that contains the current item's identifier (the map key or set value).

each.value - A special value available inside a `for_each` block that contains the current item's data. For maps, this is the value associated with the key. For sets, this equals `each.key`.

for expression - A way to transform one collection into another. Format: `[for item in collection : expression]` produces a list, `{for item in collection : key => value}` produces a map.

for_each - A meta-argument that creates one copy of a resource for each item in a map or set. Each copy is identified by its key, not a numeric index.

Map - A collection where you can add any keys you want, and each key points to a value. Like a contact list where you add names and each name has associated information. All values must be the same type. Example: `{ "web-1" = "t3.nano", "db-1" = "t3.micro" }`.

Map of Objects - A map where each value is an object. Combines flexible keys (add as many as you want) with structured values (each has the same fields). Used to define complex configuration data where each item has multiple properties.

Object - A structure with fixed, named fields where each field can have a different type. Like a form with specific blanks to fill in. Example: `{ instance_type = "t3.nano", tier = "web", port = 80 }`.

Set - A collection of unique values with no particular order. Unlike lists, sets cannot contain duplicates. Created with `toset()` or set literals.

toset() - A function that converts a list to a set, removing any duplicate values in the process.

contains() - A function that checks if a list or set includes a specific value. Returns `true` if found, `false` otherwise. Syntax: `contains(collection, value)`.

Documentation Links

- for_each Meta-Argument
- for Expressions
- Type Constraints - map
- toset Function
- contains Function