# Module 1: Conditional Resource Creation

## Learning Objectives

By the end of this module you will understand:

- How to make Terraform choose between two options automatically
- How to use feature flags to create optional resources only when needed
- How to safely display information about resources that might not exist

---

## Why Do We Need Conditional Logic?

Imagine you're setting up infrastructure for two environments: dev and staging.

Each environment has different needs:

- **dev** is for developers to test code quickly. It doesn't need a database server, uses the smallest (cheapest) instance size, and has small storage.
- **staging** is for testing before release. It needs a database server, uses a larger instance, and has more storage.

**The Problem**: With basic variables and tfvars, you can parameterize values - but what if you need to skip creating a resource entirely in some environments? Or choose between two completely different values based on a condition?

**The Solution**: Conditional expressions let your configuration make decisions automatically - creating or skipping resources, and selecting values based on conditions.

---

## Quick Note: Locals

Before we dive into conditionals, you'll see `local.` references in the examples below. Locals are internal values you define once and reuse throughout your configuration.

```
locals {
  environment = terraform.workspace    # Gets the current workspace name
  name_prefix = "myapp-${local.environment}"
}
```

- **Define** them in a `locals` block
- **Reference** them with `local.` prefix (e.g., `local.environment`)

Think of locals as a way to calculate a value once and give it a name you can use everywhere.

---

## The Ternary Operator (Inline If-Else)

The ternary operator lets Terraform pick between two options based on a true/false condition.

### The Format

```
condition ? value_if_condition_is_true : value_if_condition_is_false
```

### What Counts as True or False?

In Terraform, conditions evaluate to either `true` or `false`:

- **Boolean values**: `true` is true, `false` is false
- **Comparisons**: `var.environment == "staging"` is true only when environment equals "staging"
- **Not equal**: `var.environment != "dev"` is true when environment is anything other than "dev"

**Example**: Choose an instance type based on whether we're in staging:

```
instance_type = var.environment == "staging" ? "t3.small" : "t3.nano"
```

Let's break down each part:

- `var.environment == "staging"` - This is the **condition** (true when environment equals "staging")
- `?` - This symbol means "if true, use the next value"
- `"t3.small"` - This is the value used **when the condition is true**
- `:` - This symbol means "otherwise, use this instead"
- `"t3.nano"` - This is the value used **when the condition is false**

This says: "If environment equals staging, use t3.small. Otherwise, use t3.nano."

**Choosing storage size:**

```
volume_size = var.environment == "staging" ? 20 : 10
```

This says: "If environment equals staging, use 20 GB. Otherwise, use 10 GB."

---

## Using count to Create or Skip Resources

Sometimes you don't just want to choose between two values - you want to **create a resource or skip it entirely**.

The `count` setting controls how many copies of a resource Terraform creates:

- `count = 1` means "create this resource" (one copy)
- `count = 0` means "don't create this resource at all" (zero copies)

### The Pattern

We combine `count` with the ternary operator:

```
resource "aws_instance" "db" {
  count = var.enable_db ? 1 : 0

  ami           = data.aws_ami.amazon_linux.id
  instance_type = "t3.micro"
}
```

This says: "If enable_db is true, create 1 instance. If false, create 0 instances (skip it)."

### How It Works Step by Step

**When `var.enable_db = true`:**

1. Terraform evaluates `true ? 1 : 0`
2. The condition is true, so it picks the first value: `1`
3. `count = 1` tells Terraform to create the resource

**When `var.enable_db = false`:**

1. Terraform evaluates `false ? 1 : 0`
2. The condition is false, so it picks the second value: `0`
3. `count = 0` tells Terraform to skip this resource completely

### Referencing Resources That Use count

When you add `count` to a resource, Terraform prepares to potentially create multiple copies (0, 1, 2, or more). To handle this, it stores the resource as a list. Even with `count = 1`, you get a list containing one item - that's why you need `[0]` to access it:

```
security_group_id = aws_security_group.db[0].id
```

The `[0]` means "get the first item in the list" (lists start counting at 0, not 1).

**Warning**: If `count = 0`, there is no first item - the list is empty! Trying to access `[0]` would cause an error. We'll learn how to handle this safely in the "Conditional Outputs" section.

---

## Feature Flags: On/Off Switches for Resources

A feature flag is simply a true/false variable that acts like a light switch - it turns a feature on or off.

### Creating a Feature Flag

```
variable "enable_db_instance" {
  description = "Enable optional database EC2 instance"
  type        = bool
  default     = false
}
```

This creates an on/off switch called `enable_db_instance`. By default, it's off (`false`).

### Using Feature Flags

You can use the same feature flag to control multiple related resources:

```
# Security group - only created when DB is enabled
resource "aws_security_group" "db_sg" {
  count = var.enable_db_instance ? 1 : 0

  name        = "${local.name_prefix}-db-sg"
  description = "Security group for database server"
}

# DB instance - only created when DB is enabled
resource "aws_instance" "db" {
  count = var.enable_db_instance ? 1 : 0

  ami                    = data.aws_ami.amazon_linux.id
  instance_type          = "t3.micro"
  vpc_security_group_ids = [aws_security_group.db_sg[0].id]
}
```

Both the security group and the database instance use the same condition (`count = var.enable_db_instance ? 1 : 0`), so they're always created together or skipped together.

**Important:** When a resource uses `count = 1`, you must reference it with `[0]` to access the first (and only) instance:

```
vpc_security_group_ids = [aws_security_group.db_sg[0].id]
```

### Setting Feature Flags in tfvars Files

You can set different values for different environments:

**dev.tfvars:**

```
enable_db_instance = false
```

Dev doesn't need a database, so the switch is off.

**staging.tfvars:**

```
enable_db_instance = true
```

Staging needs a database, so the switch is on.

### Overriding from the Command Line

You can also flip the switch temporarily when running Terraform:

```
terraform plan -var-file="dev.tfvars" -var="enable_db_instance=true"
```

This uses dev settings but overrides the database switch to "on" - useful for testing.

---

## Choosing Values Based on Environment

You can use the ternary operator directly inside resource definitions to pick different values:

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

  # Pick instance size based on environment
  instance_type = local.environment == "staging" ? "t3.small" : "t3.nano"

  root_block_device {
    # Pick storage size based on environment
    volume_size = local.environment == "staging" ? 20 : 10
    volume_type = "gp3"
    encrypted   = true
  }
}
```

### Another Example: Security Rules

```
resource "aws_vpc_security_group_ingress_rule" "http" {
  security_group_id = aws_security_group.web.id
  from_port         = 80
  to_port           = 80
  ip_protocol       = "tcp"

  # Dev: allow access from anywhere (for easy testing)
  # Other environments: only allow internal network access (more secure)
```

```
  cidr_ipv4 = local.environment == "dev" ? "0.0.0.0/0" : "10.0.0.0/8"
}
```

---

## Conditional Outputs: Safely Showing Information

Outputs display information after Terraform runs. But what if you want to show information about a resource that might not exist?

### The Problem

If a resource wasn't created (because `count = 0`), trying to access it causes an error:

```
# This FAILS when the database wasn't created
output "db_instance_id" {
  value = aws_instance.db[0].id    # Error! There's no [0] because the
list is empty
}
```

### The Solution

Use the same condition that controls whether the resource exists:

```
output "db_instance_id" {
  description = "DB instance ID (shows null if database wasn't created)"
  value       = var.enable_db_instance ? aws_instance.db[0].id : null
}
```

This says: "If the database is enabled, show the ID. Otherwise, show null (nothing)."

**How it works:**

- If `enable_db_instance = true`, the database exists, so it's safe to access `[0]`
- If `enable_db_instance = false`, the database doesn't exist, so we show `null` instead of causing an error

---

## Key Takeaways

1. **The ternary operator** (`condition ? value_if_true : value_if_false`) lets Terraform choose between two options automatically
2. **count = 0 or 1** controls whether a resource is created or skipped completely
3. **Feature flags** are on/off switches (true/false variables) that control optional resources

4. **Conditional outputs** must use the same condition as the resource to avoid errors when the resource doesn't exist
5. **Related resources** (like a database and its security group) should use the same switch so they're always created or skipped together

---

## Knowledge Check

### Question 1

What does this expression give you when `var.environment = "dev"`?

```
instance_type = var.environment == "staging" ? "t3.small" : "t3.nano"
```

A) `"t3.small"` B) `"t3.nano"` C) `"dev"` D) Error

---

### Question 2

What happens when you set `count = 0` on a resource?

A) Creates a zero-sized resource B) Creates one resource with empty values C) Terraform shows an error D) The resource is not created at all

---

### Question 3

Why must conditional outputs use the same condition as the resource?

```
resource "aws_instance" "db" {
  count = var.enable_db ? 1 : 0
  # ...
}

output "db_id" {
  value = var.enable_db ? aws_instance.db[0].id : null
}
```

A) To make the code easier to read B) Because `[0]` doesn't exist when `count = 0`, which would cause an error C) Because Terraform requires it D) To make Terraform run faster

---

### Answers

**Question 1: B** - The condition `var.environment == "staging"` is false (dev does not equal staging), so Terraform picks the second value: `"t3.nano"`.

**Question 2: D** - `count = 0` means "create zero copies" - the resource is completely skipped and won't exist.

**Question 3: B** - When `count = 0`, the resource list is empty (nothing was created). Trying to get `[0]` (the first item) from an empty list causes an error. The ternary operator checks first, so we only try to access `[0]` when we know the resource exists.

---

## Dictionary of Terms

**Boolean** - A value that can only be `true` or `false`. Example: `enable_db_instance = true`

**Conditional Expression** - Code that produces different results based on a true/false condition. In Terraform, the ternary operator is the main way to write conditional expressions.

**count** - A setting that tells Terraform how many copies of a resource to create. Use `count = 1` to create the resource, or `count = 0` to skip it entirely.

**Feature Flag** - A true/false variable used as an on/off switch to control whether certain resources are created.

**Index** - A number that identifies a specific item in a list. Lists start at 0, so `[0]` means "the first item". When using `count`, you access resources using their index: `resource[0]`.

**Locals** - Internal values defined in a `locals` block. Use them to calculate a value once and reuse it throughout your configuration. Referenced with `local.` prefix (e.g., `local.environment`).

**Meta-argument** - A special setting (like `count`) that works with any resource type and changes how Terraform handles that resource.

**null** - A special value meaning "nothing" or "no value". Used in conditional outputs when a resource doesn't exist.

**Ternary Operator** - A conditional expression with the format: `condition ? value_if_true : value_if_false`. Picks one of two values based on whether the condition is true or false.

**tfvars File** - A file containing variable values for a specific situation (like `dev.tfvars` or `staging.tfvars`). Loaded using the `-var-file` flag.

---

## Documentation Links

- count Meta-Argument
- Conditional Expressions