EXAMPLE CODE

As a reminder, you can find all of the code examples in the book on GitHub.

Loops

Terraform offers several different looping constructs, each intended to be used in a slightly different scenario:

- `count` parameter, to loop over resources and modules
- `for_each` expressions, to loop over resources, inline blocks within a resource, and modules
- `for` expressions, to loop over lists and maps
- `for` string directive, to loop over lists and maps within a string

Let's go through these one at a time.

Loops with the count Parameter

In Chapter 2, you created an AWS Identity and Access Management (IAM) user by clicking around the Console. Now that you have this user, you can create and manage all future IAM users with Terraform. Consider the following Terraform code, which should live in *live/global/iam/main.tf*:

```
provider "aws" {
  region = "us-east-2"
}

resource "aws_iam_user" "example" {
  name = "neo"
}
```

This code uses the `aws_iam_user` resource to create a single new IAM user. What if you want to create three IAM users? In a general-purpose programming language, you'd probably use a for-loop:

```
# This is just pseudo code. It won't actually work in Terraform.
for (i = 0; i < 3; i++) {
  resource "aws_iam_user" "example" {
    name = "neo"
  }
}
```

Terraform does not have for-loops or other traditional procedural logic built into the language, so this syntax will not work. However, every Terraform resource has a meta-parameter you can use called `count`. `count` is Terraform's oldest, simplest, and most limited iteration construct: all it does is define how many copies of the resource to create. Here's how you use `count` to create three IAM users:

```
resource "aws_iam_user" "example" {
  count = 3
  name  = "neo"
}
```

One problem with this code is that all three IAM users would have the same name, which would cause an error, since usernames must be unique. If you had access to a standard for-loop, you might use the index in the for-loop, `i`, to give each user a unique name:

```
# This is just pseudo code. It won't actually work in Terraform.
for (i = 0; i < 3; i++) {
  resource "aws_iam_user" "example" {
    name = "neo.${i}"
  }
}
```

To accomplish the same thing in Terraform, you can use `count.index` to get the index of each "iteration" in the "loop":

```
resource "aws_iam_user" "example" {
  count = 3
  name  = "neo.${count.index}"
}
```

If you run the `plan` command on the preceding code, you will see that Terraform wants to create three IAM users, each with a different name (`"neo.0"`, `"neo.1"`, `"neo.2"`):

```
Terraform will perform the following actions:

  # aws_iam_user.example[0] will be created
  + resource "aws_iam_user" "example" {
      + name          = "neo.0"
      (...)
    }

  # aws_iam_user.example[1] will be created
  + resource "aws_iam_user" "example" {
      + name          = "neo.1"
      (...)
    }

  # aws_iam_user.example[2] will be created
  + resource "aws_iam_user" "example" {
      + name          = "neo.2"
      (...)
    }

Plan: 3 to add, 0 to change, 0 to destroy.
```

Of course, a username like `"neo.0"` isn't particularly usable. If you combine `count.index` with some built-in functions from Terraform, you can customize each "iteration" of the "loop" even more.

For example, you could define all of the IAM usernames you want in an input variable in *live/global/iam/variables.tf*:

```
variable "user_names" {
  description = "Create IAM users with these names"
  type        = list(string)
  default     = ["neo", "trinity", "morpheus"]
}
```

If you were using a general-purpose programming language with loops and arrays, you would configure each IAM user to use a different name by looking up index `i` in the array `var.user_names`:

```
# This is just pseudo code. It won't actually work in Terraform.
for (i = 0; i < 3; i++) {
  resource "aws_iam_user" "example" {
    name = vars.user_names[i]
  }
}
```

In Terraform, you can accomplish the same thing by using `count` along with the following:

Array lookup syntax

The syntax for looking up members of an array in Terraform is similar to most other programming languages:

```
ARRAY[<INDEX>]
```

For example, here's how you would look up the element at index 1 of `var.user_names`:

```
var.user_names[1]
```

The `length` function

Terraform has a built-in function called `length` that has the following syntax:

```
length(<ARRAY>)
```

As you can probably guess, the `length` function returns the number of items in the given `ARRAY`. It also works with strings and maps.

Putting these together, you get the following:

```
resource "aws_iam_user" "example" {
  count = length(var.user_names)
```

```
    name  = var.user_names[count.index]
  }
```

Now when you run the `plan` command, you'll see that Terraform wants to create three IAM users, each with a unique, readable name:

```
Terraform will perform the following actions:

  # aws_iam_user.example[0] will be created
  + resource "aws_iam_user" "example" {
      + name          = "neo"
      (...)
    }

  # aws_iam_user.example[1] will be created
  + resource "aws_iam_user" "example" {
      + name          = "trinity"
      (...)
    }

  # aws_iam_user.example[2] will be created
  + resource "aws_iam_user" "example" {
      + name          = "morpheus"
      (...)
    }

Plan: 3 to add, 0 to change, 0 to destroy.
```

Note that after you've used `count` on a resource, it becomes an array of resources rather than just one resource. Because `aws_iam_user.example` is now an array of IAM users, instead of using the standard syntax to read an attribute from that resource (`<PROVIDER>_<TYPE>.<NAME>.<ATTRIBUTE>`), you must specify which IAM user you're interested in by specifying its index in the array using the same array lookup syntax:

```
<PROVIDER>_<TYPE>.<NAME>[INDEX].ATTRIBUTE
```

For example, if you want to provide the Amazon Resource Name (ARN) of the first IAM user in the list as an output variable, you would need to do the following:

```
output "first_arn" {
  value       = aws_iam_user.example[0].arn
  description = "The ARN for the first user"
}
```

If you want the ARNs of *all* of the IAM users, you need to use a *splat expression*, "*", instead of the index:

```
output "all_arns" {
  value       = aws_iam_user.example[*].arn
  description = "The ARNs for all users"
}
```

When you run the `apply` command, the `first_arn` output will contain just the ARN for `neo`, whereas the `all_arns` output will contain the list of all ARNs:

```
$ terraform apply

(...)

Apply complete! Resources: 3 added, 0 changed, 0 destroyed.

Outputs:

first_arn = "arn:aws:iam::123456789012:user/neo"
all_arns = [
  "arn:aws:iam::123456789012:user/neo",
  "arn:aws:iam::123456789012:user/trinity",
  "arn:aws:iam::123456789012:user/morpheus",
]
```

As of Terraform 0.13, the `count` parameter can also be used on modules. For example, imagine you had a module at *modules/landing-zone/iam-user* that can create a single IAM user:

```
resource "aws_iam_user" "example" {
  name = var.user_name
}
```

The username is passed into this module as an input variable:

```
variable "user_name" {
  description = "The user name to use"
  type        = string
}
```

And the module returns the ARN of the created IAM user as an output variable:

```
output "user_arn" {
  value       = aws_iam_user.example.arn
  description = "The ARN of the created IAM user"
}
```

You could use this module with a `count` parameter to create three IAM users as follows:

```
module "users" {
  source = "../../../modules/landing-zone/iam-user"

  count     = length(var.user_names)
  user_name = var.user_names[count.index]
}
```

The preceding code uses `count` to loop over this list of usernames:

```
variable "user_names" {
  description = "Create IAM users with these names"
  type        = list(string)
  default     = ["neo", "trinity", "morpheus"]
}
```

And it outputs the ARNs of the created IAM users as follows:

```
output "user_arns" {
  value       = module.users[*].user_arn
  description = "The ARNs of the created IAM users"
}
```

Just as adding `count` to a resource turns it into an array of resources, adding `count` to a module turns it into an array of modules.

If you run `apply` on this code, you'll get the following output:

```
$ terraform apply

(...)

Apply complete! Resources: 3 added, 0 changed, 0 destroyed.

Outputs:

all_arns = [
  "arn:aws:iam::123456789012:user/neo",
  "arn:aws:iam::123456789012:user/trinity",
  "arn:aws:iam::123456789012:user/morpheus",
]
```

So, as you can see, `count` works more or less identically with resources and with modules.

Unfortunately, `count` has two limitations that significantly reduce its usefulness. First, although you can use `count` to loop over an entire resource, you can't use `count` within a resource to loop over inline blocks.

For example, consider how tags are set in the `aws_autoscaling_group` resource:

```
resource "aws_autoscaling_group" "example" {
  launch_configuration = aws_launch_configuration.example.name
  vpc_zone_identifier  = data.aws_subnets.default.ids
  target_group_arns    = [aws_lb_target_group.asg.arn]
  health_check_type    = "ELB"

  min_size = var.min_size
  max_size = var.max_size

  tag {
    key                 = "Name"
    value               = var.cluster_name
    propagate_at_launch = true
  }
}
```

Each `tag` requires you to create a new inline block with values for `key`, `value`, and `propagate_at_launch`. The preceding code hardcodes a single tag, but you might want to allow users to pass in custom tags. You might be tempted to try to use the `count` parameter to loop over these tags and generate dynamic inline `tag` blocks, but unfortunately, using `count` within an inline block is not supported.

The second limitation with `count` is what happens when you try to change its value. Consider the list of IAM users you created earlier:

```
variable "user_names" {
  description = "Create IAM users with these names"
  type        = list(string)
  default     = ["neo", "trinity", "morpheus"]
}
```

Imagine that you removed `"trinity"` from this list. What happens when you run `terraform plan`?

```
$ terraform plan

(...)

Terraform will perform the following actions:

  # aws_iam_user.example[1] will be updated in-place
  ~ resource "aws_iam_user" "example" {
        id             = "trinity"
      ~ name           = "trinity" -> "morpheus"
    }

  # aws_iam_user.example[2] will be destroyed
  - resource "aws_iam_user" "example" {
      - id             = "morpheus" -> null
      - name           = "morpheus" -> null
    }

Plan: 0 to add, 1 to change, 1 to destroy.
```

Wait a second, that’s probably not what you were expecting! Instead of just deleting the `"trinity"` IAM user, the `plan` output is indicating that

Terraform wants to rename the "`trinity`" IAM user to "`morpheus`" and delete the original "`morpheus`" user. What’s going on?

When you use the `count` parameter on a resource, that resource becomes an array of resources. Unfortunately, the way Terraform identifies each resource within the array is by its position (index) in that array. That is, after running `apply` the first time with three usernames, Terraform’s internal representation of these IAM users looks something like this:

```
aws_iam_user.example[0]: neo
aws_iam_user.example[1]: trinity
aws_iam_user.example[2]: morpheus
```

When you remove an item from the middle of an array, all the items after it shift back by one, so after running `plan` with just two bucket names, Terraform’s internal representation will look something like this:

```
aws_iam_user.example[0]: neo
aws_iam_user.example[1]: morpheus
```

Notice how "`morpheus`" has moved from index 2 to index 1. Because it sees the index as a resource’s identity, to Terraform, this change roughly translates to “rename the bucket at index 1 to morpheus and delete the bucket at index 2.” In other words, every time you use `count` to create a list of resources, if you remove an item from the middle of the list, Terraform will delete every resource after that item and then re-create those resources again from scratch. Ouch. The end result, of course, is exactly what you requested (i.e., two IAM users named "`morpheus`" and "`neo`"), but deleting resources is probably not how you want to get there, as you may lose availability (you can’t use the IAM user during the `apply`), and, even worse, you may lose data (if the resource you’re deleting is a database, you may lose all the data in it!).

To solve these two limitations, Terraform 0.12 introduced `for_each` expressions.