Loops with for_each Expressions

The *for_each* expression allows you to loop over lists, sets, and maps to create (a) multiple copies of an entire resource, (b) multiple copies of an inline block within a resource, or (c) multiple copies of a module. Let's first walk through how to use `for_each` to create multiple copies of a resource.

The syntax looks like this:

```
resource "<PROVIDER>_<TYPE>" "<NAME>" {
  for_each = <COLLECTION>

  [CONFIG ...]
}
```

where `COLLECTION` is a set or map to loop over (lists are not supported when using `for_each` on a resource) and `CONFIG` consists of one or more arguments that are specific to that resource. Within `CONFIG`, you can use `each.key` and `each.value` to access the key and value of the current item in `COLLECTION`.

For example, here's how you can create the same three IAM users using `for_each` on a resource:

```
resource "aws_iam_user" "example" {
  for_each = toset(var.user_names)
  name     = each.value
}
```

Note the use of `toset` to convert the `var.user_names` list into a set. This is because `for_each` supports sets and maps only when used on a resource. When `for_each` loops over this set, it makes each username available in `each.value`. The username will also be available in `each.key`, though you typically use `each.key` only with maps of key-value pairs.

Once you've used `for_each` on a resource, it becomes a map of resources, rather than just one resource (or an array of resources as with `count`). To see what that means, remove the original `all_arns` and `first_arn` output variables, and add a new `all_users` output variable:

```
output "all_users" {
  value = aws_iam_user.example
}
```

Here's what happens when you run `terraform apply`:

```
$ terraform apply

(...)

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

Outputs:

all_users = {
  "morpheus" = {
    "arn" = "arn:aws:iam::123456789012:user/morpheus"
    "force_destroy" = false
    "id" = "morpheus"
    "name" = "morpheus"
    "path" = "/"
    "tags" = {}
  }
  "neo" = {
    "arn" = "arn:aws:iam::123456789012:user/neo"
    "force_destroy" = false
    "id" = "neo"
    "name" = "neo"
    "path" = "/"
    "tags" = {}
  }
  "trinity" = {
    "arn" = "arn:aws:iam::123456789012:user/trinity"
    "force_destroy" = false
    "id" = "trinity"
    "name" = "trinity"
    "path" = "/"
    "tags" = {}
    }
  }
```

You can see that Terraform created three IAM users and that the `all_users` output variable contains a map where the keys are the keys in `for_each` (in this case, the usernames) and the values are all the outputs for that resource. If you want to bring back the `all_arns` output variable, you’d need to do a little extra work to extract those ARNs using the `values` built-in function (which returns just the values from a map) and a splat expression:

```
output "all_arns" {
  value = values(aws_iam_user.example)[*].arn
}
```

This gives you the expected output:

```
$ terraform apply

(...)

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

Outputs:

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

The fact that you now have a map of resources with `for_each` rather than an array of resources as with `count` is a big deal, because it allows you to remove items from the middle of a collection safely. For example, if you again remove `"trinity"` from the middle of the `var.user_names` list and run `terraform plan`, here’s what you’ll see:

```
$ terraform plan

Terraform will perform the following actions:
  # aws_iam_user.example["trinity"] will be destroyed
  - resource "aws_iam_user" "example" {
      - arn           = "arn:aws:iam::123456789012:user/trinity"
-> null
      - name          = "trinity" -> null
    }

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

That's more like it! You're now deleting solely the exact resource you want, without shifting all of the other ones around. This is why you should almost always prefer to use `for_each` instead of `count` to create multiple copies of a resource.

`for_each` works with modules in a more or less identical fashion. Using the `iam-user` module from earlier, you can create three IAM users with it using `for_each` as follows:

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

  for_each  = toset(var.user_names)
  user_name = each.value
}
```

And you can output the ARNs of those users as follows:

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

When you run `apply` on this code, you get the expected output:

```
$ terraform apply

(...)

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

Outputs:
```

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

Let's now turn our attention to another advantage of `for_each`: its ability to create multiple inline blocks within a resource. For example, you can use `for_each` to dynamically generate `tag` inline blocks for the ASG in the `webserver-cluster` module. First, to allow users to specify custom tags, add a new map input variable called `custom_tags` in *modules/services/webserver-cluster/variables.tf*:

```
variable "custom_tags" {
  description = "Custom tags to set on the Instances in the ASG"
  type        = map(string)
  default     = {}
}
```

Next, set some custom tags in the production environment, in *live/prod/services/webserver-cluster/main.tf*, as follows:

```
module "webserver_cluster" {
  source = "../../../../modules/services/webserver-cluster"

  cluster_name           = "webservers-prod"
  db_remote_state_bucket = "(YOUR_BUCKET_NAME)"
  db_remote_state_key    = "prod/data-
stores/mysql/terraform.tfstate"

  instance_type        = "m4.large"
  min_size             = 2
  max_size             = 10

  custom_tags = {
    Owner     = "team-foo"
    ManagedBy = "terraform"
  }
}
```

The preceding code sets a couple of useful tags: the `Owner` tag specifies which team owns this ASG, and the `ManagedBy` tag specifies that this infrastructure is managed using Terraform (indicating that this infrastructure shouldn't be modified manually).

Now that you've specified your tags, how do you actually set them on the `aws_autoscaling_group` resource? What you need is a for-loop over `var.custom_tags`, similar to the following pseudocode:

```
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
  }

  # This is just pseudo code. It won't actually work in
Terraform.
  for (tag in var.custom_tags) {
    tag {
      key                 = tag.key
      value               = tag.value
      propagate_at_launch = true
    }
  }
}
```

The preceding pseudocode won't work, but a `for_each` expression will. The syntax for using `for_each` to dynamically generate inline blocks looks like this:

```
dynamic "<VAR_NAME>" {
  for_each = <COLLECTION>

  content {
    [CONFIG...]
  }
}
```

where `VAR_NAME` is the name to use for the variable that will store the value of each "iteration," `COLLECTION` is a list or map to iterate over, and the `content` block is what to generate from each iteration. You can use `<VAR_NAME>.key` and `<VAR_NAME>.value` within the `content` block to access the key and value, respectively, of the current item in the `COLLECTION`. Note that when you're using `for_each` with a list, the `key` will be the index, and the `value` will be the item in the list at that index, and when using `for_each` with a map, the `key` and `value` will be one of the key-value pairs in the map.

Putting this all together, here is how you can dynamically generate `tag` blocks using `for_each` 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
  }

  dynamic "tag" {
    for_each = var.custom_tags

    content {
      key                 = tag.key
      value               = tag.value
      propagate_at_launch = true
    }
  }
}
```

If you run `terraform plan` now, you should see a plan that looks something like this:

```
$ terraform plan

Terraform will perform the following actions:

  # aws_autoscaling_group.example will be updated in-place
  ~ resource "aws_autoscaling_group" "example" {
        (...)

        tag {
            key                 = "Name"
            propagate_at_launch = true
            value               = "webservers-prod"
        }
      + tag {
          + key                 = "Owner"
          + propagate_at_launch = true
          + value               = "team-foo"
        }
      + tag {
          + key                 = "ManagedBy"
          + propagate_at_launch = true
          + value               = "terraform"
        }
    }

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