Terraform tips & tricks: loops, if-statements, and gotchas

Yevgeniy Brikman
Oct 10, 2016 · 31 min read

Image by Florian Richter

***Update, November 17, 2016**: We took this blog post series, expanded it, and turned it into a book called Terraform: Up & Running!*

***Update, July 8, 2019**: We've updated this blog post series for Terraform 0.12 and released the 2nd edition of Terraform: Up & Running!*

This is Part 5 of the Comprehensive Guide to Terraform series. In previous parts, we explained why we picked Terraform, introduced the basic syntax and features of

Terraform, discussed how to manage Terraform state, and showed how to create reusable infrastructure with Terraform modules. In this part, we are going to expand your Terraform toolbox with some more advanced tips & tricks, such as how to do loops and if-statements. We'll also discuss some of Terraform's weaknesses so you can avoid the most common gotchas.

Terraform is a declarative language. As discussed in Part 1 of this series, infrastructure-as-code in a declarative language tends to provides a more accurate view of what's actually deployed than a procedural language, so it's easier to reason about and makes it easier to keep the code base small. However, without access to a full programming language, certain types of tasks become more difficult in a declarative language.

For example, since declarative languages typically don't have for-loops, how do you repeat a piece of logic — such as creating multiple similar EC2 Instances — without copy and paste? And if the declarative language doesn't support if-statements, how can you conditionally configure resources, such as creating public IP addresses for frontend services, but not for backend services?

Fortunately, Terraform provides a few primitives—namely, the `count` meta-parameter, `for_each` and `for` expressions, a lifecycle block called `create_before_destroy`, a ternary operator, plus a large number of functions—that allow you to do certain types of loops, if-statements, and other logic.

Here are the topics we'll go over:

- Loops
- Conditionals
- Terraform Gotchas

You can find sample code for the examples below at: https://github.com/gruntwork-io/intro-to-terraform. Note that all the code samples are written for Terraform 0.12.x.

Loops

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

- **`count` parameter**: loop over resources.

- `for_each` **expressions**: loop over resources and inline blocks within a resource.
- `for` **expressions**: loop over lists and maps.

Let's go through these one at a time.

count parameter

In An introduction to Terraform, we created an IAM user by clicking around the AWS console. Now that you have this user, you can create and manage all future IAM users with Terraform. Consider the following Terraform code:

```
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 wanted 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`. This is Terraform's oldest, simplest, and most limited iteration construct: all it does is define how many copies of the resource to create. Therefore, you can create three IAM users as follows:

```
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 slightly 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" {
      + arn           = (known after apply)
      + force_destroy = false
      + id            = (known after apply)
      + name          = "neo.0"
      + path          = "/"
      + unique_id     = (known after apply)
    }

  # aws_iam_user.example[1] will be created
  + resource "aws_iam_user" "example" {
      + arn           = (known after apply)
      + force_destroy = false
      + id            = (known after apply)
```

```
      + name          = "neo.1"
      + path          = "/"
      + unique_id     = (known after apply)
    }

  # aws_iam_user.example[2] will be created
  + resource "aws_iam_user" "example" {
      + arn           = (known after apply)
      + force_destroy = false
      + id            = (known after apply)
      + name          = "neo.2"
      + path          = "/"
      + unique_id     = (known after apply)
    }

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

Of course, a username like “neo.0” isn’t particularly readable. 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 called `user_names`:

```
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` plus two new tricks. The first trick is to use *array lookup syntax* to look up elements in an array at a given index. This syntax is identical to what you see in most other programming languages:

```
LIST[<INDEX>]
```

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

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

The second trick is to use Terraform's built-in function `length`, which has the following syntax:

```
length(<LIST>)
```

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

Putting these together, you get:

```
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 completely different name ("neo," "trinity," "morpheus"):

```
Terraform will perform the following actions:

  # aws_iam_user.example[0] will be created
  + resource "aws_iam_user" "example" {
      + arn           = (known after apply)
      + force_destroy = false
      + id            = (known after apply)
```

```
      + name          = "neo"
      + path          = "/"
      + unique_id     = (known after apply)
    }

  # aws_iam_user.example[1] will be created
  + resource "aws_iam_user" "example" {
      + arn           = (known after apply)
      + force_destroy = false
      + id            = (known after apply)
      + name          = "trinity"
      + path          = "/"
      + unique_id     = (known after apply)
    }

  # aws_iam_user.example[2] will be created
  + resource "aws_iam_user" "example" {
      + arn           = (known after apply)
      + force_destroy = false
      + id            = (known after apply)
      + name          = "morpheus"
      + path          = "/"
      + unique_id     = (known after apply)
    }

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

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

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

For example, if you wanted to provide the Amazon Resource Name (ARN) of one of the IAM users as an output variable, you would need to do the following:

```
output "neo_arn" {
  value       = aws_iam_user.example[0].arn
  description = "The ARN for user Neo"
}
```

If you want the ARNs of *all* 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 `neo_arn` output will contain just the ARN for Neo while the `all_arns` output will contain the list of all ARNs:

```
$ 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",
]
neo_arn = arn:aws:iam::123456789012:user/neo
```

Unfortunately, `count` has two limitations that significantly reduce its usefulness. First, while you can use `count` to loop over an entire resource, you can't use `count` within a resource to loop over inline blocks. An *inline block* is an argument you set within a resource of the format:

```
resource "xxx" "yyy" {
  <NAME> {
    [CONFIG...]
  }
}
```

where `NAME` is the name of the inline block (e.g., `tag`) and `CONFIG` consists of one or more arguments that are specific to that inline block (e.g., `key` and `value`). For

example, consider how are 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_subnet_ids.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 may 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` inside of an inline block is not supported.

The second limitation with `count` is what happens when you try to change it. 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 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 a list or 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 user names, 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. Since Terraform 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 recreate 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 and modifying resources is probably not how you want to get there.

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

Loops with for_each expressions

The *`for_each` expression* allows you to loop over lists, sets, and maps to create either (a) multiple copies of an entire resource or (b) multiple copies of an inline block within a resource. 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 `PROVIDER` is the name of a provider (e.g., `aws`), `TYPE` is the type of resource to create in that provider (e.g., `instance`), `NAME` is an identifier you can use throughout the Terraform code to refer to this resource (e.g., `my_instance`), `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`:

```
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, as `for_each` only supports sets and maps when used on a resource. When `for_each` loops over this set, it will make each user name available in `each.value`. The user name will also be available in `each.key`, though you typically only use `each.key` 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 a list of resources as with `count`). To see what that means, remove the original `all_arns` and `neo_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 user names) and the values are all the outputs for that resource. If you wanted to bring back the `all_arns` output variable, you'd have 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
}
```

Which 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 a list of resources as with `count` is a big deal, as 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::1234567: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 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.

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`:

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

How do you actually set these tags on the `aws_autoscaling_group` resource? What you need is a for loop over `var.custom_tags`, similar to the following pseudo code:

```
resource "aws_autoscaling_group" "example" {
  # (...)

  # 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 pseudo code above 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 each "iteration" (instead of `each`), `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" {
  # (...)

  dynamic "tag" {
    for_each = var.custom_tags

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

Loops with for expressions

You've now seen how to loop over resources and inline blocks, but what if you need a loop to generate a single value? Imagine you wrote some Terraform

code that took in a list of names:

```
variable "names" {
  description = "A list of names"
  type        = list(string)
  default     = ["neo", "trinity", "morpheus"]
}
```

How could you convert all of these names to upper case? In a general purpose programming language, such as Python, you could write the following for-loop:

```
names = ["neo", "trinity", "morpheus"]

upper_case_names = []
for name in names:
    upper_case_names.append(name.upper())

print upper_case_names

# Prints out: ['NEO', 'TRINITY', 'MORPHEUS']
```

Python offers another way to write the exact same code in one line using a syntax known as a *list comprehension*:

```
names = ["neo", "trinity", "morpheus"]

upper_case_names = [name.upper() for name in names]

print upper_case_names

# Prints out: ['NEO', 'TRINITY', 'MORPHEUS']
```

Python also allows you to filter the resulting list by specifying a condition:

```
names = ["neo", "trinity", "morpheus"]

short_upper_case_names = [name.upper() for name in names if
len(name) < 5]

print short_upper_case_names
```

```
# Prints out: ['NEO']
```

Terraform offers very similar functionality in the form of a *`for` expression* (not to be confused with the *`for_each` expression* you saw in the previous section). The basic syntax of a `for` expression is:

```
[for <ITEM> in <LIST> : <OUTPUT>]
```

Where `LIST` is a list to loop over, `ITEM` is the local variable name to assign to each item in `LIST`, and `OUTPUT` is an expression that transforms `ITEM` in some way. For example, here is the Terraform code to convert the list of names in `var.names` to upper case:

```
variable "names" {
  description = "A list of names"
  type        = list(string)
  default     = ["neo", "trinity", "morpheus"]
}

output "upper_names" {
  value = [for name in var.names : upper(name)]
}
```

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

```
$ terraform apply

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

Outputs:

upper_names = [
  "NEO",
  "TRINITY",
  "MORPHEUS",
]
```

Just as with Python's list comprehensions, you can filter the resulting list by specifying a condition:

```
variable "names" {
  description = "A list of names"
  type        = list(string)
  default     = ["neo", "trinity", "morpheus"]
}

output "short_upper_names" {
  value = [for name in var.names : upper(name) if length(name) < 5]
}
```

Running `terraform apply` on this code gives you:

```
short_upper_names = [
  "NEO",
]
```

Terraform's for expressions also allow you to loop over a map using the following syntax:

```
[for <KEY>, <VALUE> in <MAP> : <OUTPUT>]
```

Where `MAP` is a map to loop over, `KEY` and `VALUE` are the local variable names to assign to each key-value pair in `MAP`, and `OUTPUT` is an expression that transforms `KEY` and `VALUE` in some way. Here's an example:

```
variable "hero_thousand_faces" {
  description = "map"
  type        = map(string)
  default     = {
    neo      = "hero"
    trinity  = "love interest"
    morpheus = "mentor"
  }
}

output "bios" {
  value = [for name, role in var.hero_thousand_faces : "${name} is
the ${role}"]
}
```

When you run `terraform apply` on this code, you get:

```
map_example = [
  "morpheus is the mentor",
  "neo is the hero",
  "trinity is the love interest",
]
```

You can also use for expressions to output a map rather than list using the following syntax:

```
# For looping over lists
{for <ITEM> in <LIST> : <OUTPUT_KEY> => <OUTPUT_VALUE>}

# For looping over maps
{for <KEY>, <VALUE> in <MAP> : <OUTPUT_KEY> => <OUTPUT_VALUE>}
```

The only differences are that (a) you wrap the expression in curly braces rather than square brackets and (b) rather than outputting a single value each iteration, you output a key and value, separated by an arrow. For example, here is how you can transform map to make all the keys and values upper case:

```
variable "hero_thousand_faces" {
  description = "map"
  type        = map(string)
  default     = {
    neo      = "hero"
    trinity  = "love interest"
    morpheus = "mentor"
  }
}

output "upper_roles" {
  value = {for name, role in var.hero_thousand_faces : upper(name)
=> upper(role)}
}
```

The output from running this code will be:

```
upper_roles = {
  "MORPHEUS" = "MENTOR"
```

```
    "NEO" = "HERO"
    "TRINITY" = "LOVE INTEREST"
  }
```

Conditionals

Just as Terraform offers several different ways to do loops, there are also several different ways to do conditionals, each intended to be used in a slightly different scenario:

- **`count` parameter**: conditional resources.
- **`for_each` and `for` expressions**: conditional resources and inline blocks within a resource.

Let's go through each of these one at a time.

Conditionals with the count parameter

The `count` parameter you saw earlier lets you do a basic loop. If you're clever, you can use the same mechanism to do a basic conditional. Let's start by looking at if-statements in the next section and then move on to if-else statements in the section after.

If-Statements with the count parameter

In How to create reusable infrastructure with Terraform modules, we created a Terraform module that could be used as "blueprint" for deploying web server clusters (you can find the module here). The module created an Auto Scaling Group (ASG), Classic Load Balancer (CLB), security groups, and a number of other resources. One thing the module did *not* create was the auto scaling schedule. Since you only want to scale the cluster out in production, we defined the `aws_autoscaling_schedule` resources directly in the production configurations. Is there a way we could've defined the `aws_autoscaling_schedule` resources in the `webserver-cluster` module and conditionally create them for some users of the module and not create them for others?

Let's give it a shot. The first step is to add a boolean input variable in `modules/services/webserver-cluster/variables.tf` that can be used to specify whether the module should enable auto scaling:

```
variable "enable_autoscaling" {
  description = "If set to true, enable auto scaling"
  type        = bool
}
```

Now, if you had a general-purpose programming language, you could use this input variable in an if-statement:

```
# This is just pseudo code. It won't actually work in Terraform.
if var.enable_autoscaling {
  resource "aws_autoscaling_schedule" "scale_out_business_hours" {
    scheduled_action_name  = "scale-out-during-business-hours"
    min_size               = 2
    max_size               = 10
    desired_capacity       = 10
    recurrence             = "0 9 * * *"
    autoscaling_group_name = aws_autoscaling_group.example.name
  }

  resource "aws_autoscaling_schedule" "scale_in_at_night" {
    scheduled_action_name  = "scale-in-at-night"
    min_size               = 2
    max_size               = 10
    desired_capacity       = 2
    recurrence             = "0 17 * * *"
    autoscaling_group_name = aws_autoscaling_group.example.name
  }
}
```

Terraform doesn't support if-statements, so this code won't work. However, you can accomplish the same thing by using the `count` parameter and taking advantage of two properties:

1. If you set `count` to 1 on a resource, you get one copy of that resource; if you set `count` to 0, that resource is not created at all.

2. Terraform supports *conditional expressions* of the format `<CONDITION> ? <TRUE_VAL> : <FALSE_VAL>`. This `ternary syntax`, which may be familiar to you from other programming languages, will evaluate the boolean logic in `CONDITION`, and if the result is `true`, it will return `TRUE_VAL`, and if the result is `false`, it'll return `FALSE_VAL`.

Putting these two ideas together, you can update the `webserver-cluster` module as follows:

```
resource "aws_autoscaling_schedule" "scale_out_business_hours" {
  count = var.enable_autoscaling ? 1 : 0

  scheduled_action_name  = "scale-out-during-business-hours"
  min_size               = 2
  max_size               = 10
  desired_capacity       = 10
  recurrence             = "0 9 * * *"
  autoscaling_group_name = aws_autoscaling_group.example.name
}

resource "aws_autoscaling_schedule" "scale_in_at_night" {
  count = var.enable_autoscaling ? 1 : 0

  scheduled_action_name  = "scale-in-at-night"
  min_size               = 2
  max_size               = 10
  desired_capacity       = 2
  recurrence             = "0 17 * * *"
  autoscaling_group_name = aws_autoscaling_group.example.name
}
```

If `var.enable_autoscaling` is `true`, the `count` parameter for each of the `aws_autoscaling_schedule` resources will be set to 1, so one of each will be created. If `var.enable_autoscaling` is `false`, the `count` parameter for each of the `aws_autoscaling_schedule` resources will be set to 0, so neither one will be created. This is exactly the conditional logic you want!

You can now update the usage of this module in staging (in `live/stage/services/webserver-cluster/main.tf`) to disable auto scaling by setting `enable_autoscaling` to `false`:

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

  cluster_name       = "webservers-stage"
  instance_type      = "t2.micro"
  min_size           = 2
  max_size           = 2
  enable_autoscaling = false
}
```

Similarly, you can update the usage of this module in production (in `live/prod/services/webserver-cluster/main.tf`) to enable auto scaling by setting `enable_autoscaling` to `true`:

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

  cluster_name       = "webservers-prod"
  instance_type      = "m4.large"
  min_size           = 2
  max_size           = 10
  enable_autoscaling = true
}
```

If-Else-Statements with the count parameter

Now that you know how to do an if-statement, what about an if-else-statement?

Earlier in this blog post, you created several IAM users with read-only access to EC2. Imagine that you wanted to give one of these users, neo, access to CloudWatch as well, but to allow the person applying the Terraform configurations to decide if neo got only read access or both read and write access. This is a slightly contrived example, but it makes it easy to demonstrate a simple type of if-else-statement.

Here is an IAM policy that allows read-only access to CloudWatch using the `aws_iam_policy` resource and the `aws_iam_policy_document` data source:

```
resource "aws_iam_policy" "cloudwatch_read_only" {
  name   = "cloudwatch-read-only"
  policy = data.aws_iam_policy_document.cloudwatch_read_only.json
}

data "aws_iam_policy_document" "cloudwatch_read_only" {
  statement {
    effect    = "Allow"
    actions   = [
      "cloudwatch:Describe*",
      "cloudwatch:Get*",
      "cloudwatch:List*"
    ]
    resources = ["*"]
  }
}
```

And here is an IAM policy that allows full (read and write) access to CloudWatch:

```
resource "aws_iam_policy" "cloudwatch_full_access" {
  name   = "cloudwatch-full-access"
  policy = data.aws_iam_policy_document.cloudwatch_full_access.json
}

data "aws_iam_policy_document" "cloudwatch_full_access" {
  statement {
    effect    = "Allow"
    actions   = ["cloudwatch:*"]
    resources = ["*"]
  }
}
```

The goal is to attach one of these IAM policies to neo using the `aws_iam_user_policy_attachment` resource, based on the value of a new input variable called `give_neo_cloudwatch_full_access`:

```
variable "give_neo_cloudwatch_full_access" {
  description = "If true, neo gets full access to CloudWatch"
  type        = bool
}
```

If you were using a general-purpose programming language, you might write an if-else-statement that looks like this:

```
# This is just pseudo code. It won't actually work in Terraform.
if var.give_neo_cloudwatch_full_access {
  resource "aws_iam_user_policy_attachment" "neo_cloudwatch_full" {
    user       = aws_iam_user.example[0].name
    policy_arn = aws_iam_policy.cloudwatch_full_access.arn
  }
} else {
  resource "aws_iam_user_policy_attachment" "neo_cloudwatch_read" {
    user       = aws_iam_user.example[0].name
    policy_arn = aws_iam_policy.cloudwatch_read_only.arn
  }
}
```

To do this in Terraform, you can use the `count` parameter and a conditional expression on each of the resources:

```
resource "aws_iam_user_policy_attachment" "neo_cloudwatch_full" {
  count = var.give_neo_cloudwatch_full_access ? 1 : 0

  user       = aws_iam_user.example[0].name
  policy_arn = aws_iam_policy.cloudwatch_full_access.arn
}

resource "aws_iam_user_policy_attachment" "neo_cloudwatch_read" {
  count = var.give_neo_cloudwatch_full_access ? 0 : 1

  user       = aws_iam_user.example[0].name
  policy_arn = aws_iam_policy.cloudwatch_read_only.arn
}
```

This code contains two `aws_iam_user_policy_attachment` resources. The first one, which attaches the CloudWatch full access permissions, has a conditional expression that will evaluate to 1 if `var.give_neo_cloudwatch_full_access` is `true` and 0 otherwise (this is the if-clause). The second one, which attaches the CloudWatch read-only permissions, has a conditional expression that does the exact opposite, evaluating to 0 if `var.give_neo_cloudwatch_full_access` is `true` and 1 otherwise (this is the else-clause).

Using `count` and built-in functions to simulate if-else-statements is a bit of a hack, but it's one that works fairly well, and as you can see from the code, it allows you to conceal lots of complexity from your users so that they get to work with a clean and simple API.

Conditionals with for_each and for expressions

Now that you understand how to do conditional logic with resources using the `count` parameter, you can probably guess that you can use a similar strategy to do conditional logic by using a `for_each` expression. If you pass a `for_each` expression an empty collection, it will produce 0 resources or inline blocks; if you pass it a non-empty collection, it'll create one or more resources or inline blocks. The only question is, how can you conditionally decide of the collection should be empty or not?

The answer is to combine the `for_each` expression with the `for` expression! For example, recall the way you used `for_each` to set tags:

```
dynamic "tag" {
  for_each = var.custom_tags
```

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

If `var.custom_tags` is empty, then the `for_each` expression will have nothing to loop over, so no tags will be set. In other words, you already have some conditional logic here. But you can go even further, by combining the `for_each` expression with a `for` expression as follows:

```
dynamic "tag" {
  for_each = {
    for key, value in var.custom_tags:
    key => upper(value)
    if key != "Name"
  }

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

The nested `for` expression loops over `var.custom_tags`, converts each value to upper case (e.g., perhaps for consistency), and uses a conditional in the `for` expression to filter out any `key` set to `Name`. By filtering values in the `for` expression, you can implement arbitrary conditional logic.

Note that while you should almost always prefer `for_each` over `count` for creating multiple copies of a resource, when it comes to conditional logic, setting `count` to 0 or 1 tends to be simpler than setting `for_each` to an empty or non empty collection. Therefore, you may wish to use `count` to conditionally create resources, but `for_each` for all other types of loops and conditionals.

As a reminder, the sample code for the examples above is available at: https://github.com/gruntwork-io/intro-to-terraform.

Terraform Gotchas

After going through all these tips and tricks, it's worth taking a step back and pointing out a few gotchas, including those related to loops and conditionals, as well as those related to more general problems that affect Terraform as a whole:

- Count and `for_each` have limitations
- Gotchas with secrets management
- Valid plans can fail
- Refactoring can be tricky
- Eventual consistency is consistent…eventually

count and for_each Have Limitations

In the examples in this blog post, you made extensive use of the `count` parameter and `for_each` expressions in loops and if-statements. This works well, but there are two important limitations you need to be aware of:

1. You cannot reference any resource outputs in `count` or `for_each`.
2. You cannot use `count` or `for_each` within a `module` configuration.

Let's dig into these one at a time.

You cannot reference any resource outputs in `count` or `for_each`.

Imagine you wanted to deploy multiple EC2 Instances, and for some reason you didn't want to use an Auto Scaling Group. The code might look like this:

```
resource "aws_instance" "example_1" {
  count         = 3
  ami           = "ami-0c55b159cbfafe1f0"
  instance_type = "t2.micro"
}
```

Since `count` is being set to a hard-coded value, this code will work without issues, and when you run `apply`, it will create 3 EC2 Instances. Now, what if you wanted to deploy one EC2 Instance per availability zone (AZ) in the current AWS region? You could update your code to fetch the list of AZs using the `aws_availability_zones` data source

and use the `count` parameter and array lookups to "loop" over each AZ and create an EC2 Instance in it:

```
resource "aws_instance" "example_2" {
  count             = length(data.aws_availability_zones.all.names)
  availability_zone =
    data.aws_availability_zones.all.names[count.index]
  ami               = "ami-0c55b159cbfafe1f0"
  instance_type     = "t2.micro"
}

data "aws_availability_zones" "all" {}
```

Again, this code will work just fine, as `count` can reference data sources without problems. However, what happens if the number of instances you needed to create depended on the output of some resource? The easiest way to experiment with this is to use the `random_integer` resource, which, as you can probably guess from the name, returns a random integer:

```
resource "random_integer" "num_instances" {
  min = 1
  max = 3
}
```

This code generates a random integer between 1 and 3. Let's see what happens if you try to use the `result` output from this resource in the `count` parameter of your `aws_instance` resource:

```
resource "aws_instance" "example_3" {
  count         = random_integer.num_instances.result
  ami           = "ami-0c55b159cbfafe1f0"
  instance_type = "t2.micro"
}
```

If you run `terraform plan` on this code, you'll get the following error:

```
Error: Invalid count argument
```

```
on main.tf line 30, in resource "aws_instance" "example_3":
  30:   count         = random_integer.num_instances.result

The "count" value depends on resource attributes that cannot be
determined until apply, so Terraform cannot predict how many
instances will be created. To work around this, use the -target
argument to first apply only the resources that the count depends
on.
```

Terraform requires that it can compute `count` and `for_each` during the `plan` phase, *before* any resources are created or modified. That means `count` and `for_each` can reference hard-coded values, variables, data sources, and even lists of resources (so long as the length of the list can be determined during `plan`), but not computed resource outputs.

You cannot use `count` or `for_each` within a `module` configuration.

Something you may be tempted to try is to use `count` or `for_each` within a `module` configuration:

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

  count = 3

  cluster_name  = "terraform-up-and-running-example"
  server_port   = 8080
  instance_type = "t2.micro"
}
```

This code tries to use the `count` parameter on a `module` to create 3 copies of the `webserver-cluster` resources. Or, you may sometimes be tempted to try to set `count` to 0 on a `module` as a way to optionally include it or not based on some boolean condition. While the code looks perfectly reasonable, if you run `terraform plan`, you'll get the following error:

```
Error: Reserved argument name in module block

on main.tf line 13, in module "count_example":
  13:   count = 3
```

```
The name "count" is reserved for use in a future version of
Terraform.
```

Unfortunately, as of Terraform 0.12.6, using `count` or `for_each` on `module` is not supported. According to the Terraform 0.12 release notes, this is something HashiCorp plans to add in the future, so depending on when you're reading this blog post, check the Terraform CHANGELOG to see if it finally got added.

Gotchas with secrets management

One of the most common questions we get is how to handle secrets, such as passwords, API keys, and other sensitive data, with Terraform? For example, here's a snippet of Terraform code that can be used to deploy MySQL using Amazon RDS:

```
resource "aws_db_instance" "example" {
  engine               = "mysql"
  engine_version       = "5.7"
  instance_class       = "db.t2.micro"
  name                 = "example"

  # How should you manage the credentials for the master user?
  username = "???"
  password = "???"
}
```

Notice how Terraform requires you to set two secrets, `username` and `password`, which are the credentials for the master user of the database. Handling these secrets securely, and avoiding the massive security risk of having them stored in plain text in your version control system, is fairly complicated, so check out our dedicated blog post, A comprehensive guide to managing secrets in your Terraform code, to learn all the details, including how to use encrypted backends for Terraform state, environment variables, password managers, encrypted files (e.g., with KMS), secret stores (e.g., Vault, AWS Secrets Manager), and more.

Valid Plans Can Fail

Sometimes, you run the `plan` command and it shows you a perfectly valid-looking plan, but when you run `apply`, you'll get an error. For example, try to add an `aws_iam_user` resource with the exact same name you used for the IAM user you created in An introduction to Terraform:

```
resource "aws_iam_user" "existing_user" {
  # You should change this to the username of an IAM user that
  # already exists so you can practice using the terraform
  # import command
  name = "yevgeniy.brikman"
}
```

If you now run the `plan` command, Terraform will show you a plan that looks reasonable:

```
Terraform will perform the following actions:

  # aws_iam_user.existing_user will be created
  + resource "aws_iam_user" "existing_user" {
      + arn           = (known after apply)
      + force_destroy = false
      + id            = (known after apply)
      + name          = "yevgeniy.brikman"
      + path          = "/"
      + unique_id     = (known after apply)
    }

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

If you run the `apply` command, you'll get the following error:

```
Error: Error creating IAM User yevgeniy.brikman:
EntityAlreadyExists: User with name yevgeniy.brikman already exists.
 status code: 409, request id: 71cd0053-77ef-11e9-8831-41d1571eaa29

on main.tf line 10, in resource "aws_iam_user" "existing_user":
  10: resource "aws_iam_user" "existing_user" {
```

The problem, of course, is that an IAM user with that name already exists. This can happen not only with IAM users, but almost any resource. Perhaps someone created that resource manually or via CLI commands, but either way, some identifier is the same, and that leads to a conflict. There are many variations on this error, and Terraform newbies are often caught off-guard by them.

The key realization is that `terraform plan` only looks at resources in its Terraform state file. If you create resources *out-of-band*—such as by manually clicking around the AWS console—they will not be in Terraform's state file, and therefore, Terraform will not

take them into account when you run the `plan` command. As a result, a valid-looking plan may still fail.

There are two main lessons to take away from this:

1. **Once you start using Terraform, you should only use Terraform**: Once a part of your infrastructure is managed by Terraform, you should never make changes manually to it. Otherwise, you not only set yourself up for weird Terraform errors, but you also void many of the benefits of using infrastructure as code in the first place, as that code will no longer be an accurate representation of your infrastructure.
2. **If you have existing infrastructure, use the `import` command**: If you created infrastructure before you started using Terraform, you can use the `terraform import` command to add that infrastructure to Terraform's state file, so Terraform is aware of and can manage that infrastructure. Check out the import command documentation for details. Note that if you have a lot of existing resources that you want to import into Terraform, writing the Terraform code for them from scratch and importing them one at a time can be painful, so you may want to look into a tool such as Terraforming, which can import both code and state from an AWS account automatically.

Refactoring Can Be Tricky

A common programming practice is *refactoring*, where you restructure the internal details of an existing piece of code without changing its external behavior. The goal is to improve the readability, maintainability, and general hygiene of the code. Refactoring is an essential coding practice that you should do regularly. However, when it comes to Terraform, or any infrastructure as code tool, you have to be careful about what defines the "external behavior" of a piece of code, or you will run into unexpected problems.

For example, a common refactoring practice is to rename a variable or a function to give it a clearer name. Many IDEs even have built-in support for refactoring and can rename the variable or function for you, automatically, across the entire codebase. While such a renaming is something you might do without thinking twice in a general-purpose programming language, you have to be very careful in how you do it in Terraform, or it could lead to an outage.

For example, consider the `webserver-cluster` module, which has an input variable named `cluster_name`:

```
variable "cluster_name" {
  description = "The name to use for all the cluster resources"
  type        = string
}
```

Perhaps you start using this module for deploying microservices, and initially, you set your microservice's name to `foo`. Later on, you decide you want to rename the service to `bar`. This may seem like a trivial change, but it may actually cause an outage.

That's because the `webserver-cluster` module happens to use the `cluster_name` variable in a number of resources, including the `name` parameters of the two security groups and the load balancer:

```
resource "aws_elb" "example" {
  name               = var.cluster_name
  # (...)
}
```

If you change the `name` parameter of certain resources, Terraform will delete the old version of the resource and create a new version to replace it. If the resource you are deleting happens to be a load balancer, there will be nothing to route traffic to your web server cluster until the new load balancer boots up. Similarly, if the resource you are deleting happens to be a security group, your servers will reject all network traffic until the new security group is created.

Another refactor you may be tempted to do is to change a Terraform identifier. For example, consider the `aws_security_group` resource in the `webserver-cluster` module:

```
resource "aws_security_group" "instance" {
  # (...)
}
```

The identifier for this resource is called `instance`. Perhaps you were doing a refactor and you thought it would be clearer to change this name to `cluster_instance`:

```
resource "aws_security_group" "cluster_instance" {
  # (...)
}
```

What's the result? Yup, you guessed it: downtime.

Terraform associates each resource identifier with an identifier from the cloud provider, such as associating an `iam_user` resource with an AWS IAM User ID or an `aws_instance` resource with an AWS EC2 Instance ID. If you change the resource identifier, such as changing the `aws_security_group` identifier from `instance` to `cluster_instance`, then as far as Terraform knows, you deleted the old resource and have added a completely new one. As a result, if you `apply` these changes, Terraform will delete the old security group and create a new one, and in the time period in between, your servers will reject all network traffic.

There are four main lessons you should take away from this discussion:

1. **Always use the `plan` command**: All of these gotchas can be caught by running the `plan` command, carefully scanning the output, and noticing that Terraform plans to delete a resource that you probably don't want deleted.
2. **Create before destroy**: If you do want to replace a resource, then think carefully about whether its replacement should be created before you delete the original. If so, then you may be able to use the `create_before_destroy` lifecycle setting to make that happen (we first introduced `create_before_destroy` in An introduction to Terraform). Alternatively, you can also accomplish the same effect through two manual steps: first, add the new resource to your configurations and run the `apply` command; second, remove the old resource from your configurations and run the `apply` command again.
3. **Changing identifiers requires changing state**: If you want to change the identifier associated with a resource (e.g., rename an `aws_security_group` from `instance` to `cluster_instance`) without accidentally deleting and recreating that resource, you'll need to update the Terraform state accordingly. You should never

update Terraform state files by hand — instead, use the `terraform state` commands, especially `terraform state mv`, to do it for you.

4. **Some parameters are immutable**: The parameters of many resources are immutable, so if you change them, Terraform will delete the old resource and create a new one to replace it. The documentation for each resource often specifies what happens if you change a parameter, so RTFM. And, once again, make sure to always use the `plan` command, and consider whether you should use a `create_before_destroy` strategy.

Eventual Consistency Is Consistent...Eventually

The APIs for some cloud providers, such as AWS, are asynchronous and eventually consistent. *Asynchronous* means the API may send a response immediately, without waiting for the requested action to complete. *Eventually consistent* means it takes time for a change to propagate throughout the entire system, so for some period of time, you may get inconsistent responses depending on which data store replica happens to respond to your API calls.

For example, let's say you make an API call to AWS asking it to create an EC2 Instance. The API will return a "success" (i.e., 201 Created) response more or less instantly, without waiting for the EC2 Instance creation to complete. If you tried to connect to that EC2 Instance immediately, you'd most likely fail because AWS is still provisioning it or the Instance hasn't booted yet. Moreover, if you made another API call to fetch information about that EC2 Instance, you may get an error in return (i.e., 404 Not Found). That's because the information about that EC2 Instance may still be propagating throughout AWS, and it'll take a few seconds before it's available everywhere.

In short, whenever you use an asynchronous and eventually consistent API, you are supposed to wait and retry for a while until that action has completed and propagated. Unfortunately, the AWS SDK does not provide good tools for doing this, and Terraform used to be plagued with a number of bugs similar to #6813:

```
$ terraform apply
aws_subnet.private-persistence.2: InvalidSubnetID.NotFound:
The subnet ID 'subnet-xxxxxxx' does not exist
```

That is, you create a resource (e.g., a subnet), and then try to look up some data about that resource (e.g., the ID of the newly created subnet), and Terraform can't find it. Most of these bugs (including #6813) have been fixed, but they still crop up from time to time, especially when Terraform adds support for a new type of resource. These bugs are annoying, but fortunately, most of them are harmless. If you just rerun `terraform apply`, everything
will work fine, since by the time you rerun it, the information has propagated throughout the system.

Conclusion

Although Terraform is a declarative language, it includes a large number of tools — such as variables and modules, which we discussed in Part 4 of this series, and `count` and `for_each`, which we discussed in this blog post — that give the language a surprising amount of flexibility and expressive power. There are many permutations of the tricks shown in this blog post that we didn't cover, so let your inner-hacker go wild. OK, maybe not too wild, as someone still has to maintain this code, but just wild enough that you can create clean, beautiful APIs for your users.

In the next, and final, part of the series, we will discuss how to use Terraform as a team. This includes the basic workflow, pull requests, environments, testing, and more.

For an expanded version of this blog post series, pick up a copy of the book Terraform: Up & Running (2nd edition available now!). If you need help with Terraform, DevOps practices, or AWS at your company, feel free to reach out to us at Gruntwork.

Thanks to Josh Padnick.

AWS Cloud Computing DevOps Terraform Infrastructure As Code