Chapter 5. Terraform Tips and Tricks: Loops, If-Statements, Deployment, and Gotchas

Terraform is a declarative language. As discussed in Chapter 1, IaC in a declarative language tends to provide 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 codebase small. However, certain types of tasks are more difficult in a declarative language.

For example, because declarative languages typically don't have for-loops, how do you repeat a piece of logic—such as creating multiple similar resources—without copy and paste? And if the declarative language doesn't support if-statements, how can you conditionally configure resources, such as creating a Terraform module that can create certain resources for some users of that module but not for others? Finally, how do you express an inherently procedural idea, such as a zero-downtime deployment, in a declarative language?

Fortunately, Terraform provides a few primitives—namely, the meta-parameter `count`, `for_each` and `for` expressions, a ternary operator, a lifecycle block called `create_before_destroy`, and a large number of functions—that allow you to do certain types of loops, if-statements, and zero-downtime deployments. Here are the topics I'll cover in this chapter:

- Loops
- Conditionals
- Zero-downtime deployment
- Terraform gotchas

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.

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.
```

ENFORCING TAGGING STANDARDS

It's typically a good idea to come up with a tagging standard for your team and create Terraform modules that enforce this standard as code. One way to do this is to manually ensure that every resource in every module sets the proper tags, but with many resources, this is tedious and error prone. If there are tags that you want to apply to *all* of your AWS resources, a more reliable approach is to add the `default_tags` block to the `aws` provider in every one of your modules:

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

  # Tags to apply to all AWS resources by default
  default_tags {
    tags = {
      Owner     = "team-foo"
      ManagedBy = "Terraform"
    }
  }
}
```

The preceding code will ensure that every single AWS resource you create in this module will include the `Owner` and `ManagedBy` tags (the only exceptions are resources that don't support tags and the `aws_autoscaling_group` resource, which does support tags but doesn't work with `default_tags`, which is why you had to do all that work in the previous section to set tags in the `webserver-cluster` module). `default_tags` gives you a way to ensure all resources have a common baseline of tags while still allowing you to override those tags on a resource-by-resource basis. In Chapter 9, you'll see how to define and enforce policies as code such as "all resources must have a `ManagedBy` tag" using tools such as OPA.

Loops with for Expressions

You've now seen how to use loops to create multiple copies of entire resources and inline blocks, but what if you need a loop to set a single variable or parameter?

Imagine that 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 uppercase? 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 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 as follows:

```
[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 uppercase:

```
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:

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

Running `terraform apply` on this code gives you this:

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

Terraform's `for` expression also allows you to loop over a map using the following syntax:

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

Here, `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 ${rol
  }
```

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

```
  bios = [
    "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 a list using the following syntax:

```
  # Loop over a list and output a map
  {for <ITEM> in <LIST> : <OUTPUT_KEY> => <OUTPUT_VALUE>}

  # Loop over a map and output a map
  {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 a map to make all the keys and values uppercase:

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

Here's the output from running this code:

```
  upper_roles = {
    "MORPHEUS" = "MENTOR"
    "NEO" = "HERO"
    "TRINITY" = "LOVE INTEREST"
  }
```

Loops with the for String Directive

Earlier in the book, you learned about string interpolations, which allow you to reference Terraform code within strings:

```
"Hello, ${var.name}"
```

String directives allow you to use control statements (e.g., for-loops and if-statements) within strings using a syntax similar to string interpolations, but instead of a dollar sign and curly braces (`${…}`), you use a percent sign and curly braces (`%{…}`).

Terraform supports two types of string directives: for-loops and conditionals. In this section, we'll go over for-loops; we'll come back to conditionals later in the chapter. The `for` string directive uses the following syntax:

```
%{ for <ITEM> in <COLLECTION> }<BODY>%{ endfor }
```

where `COLLECTION` is a list or map to loop over, `ITEM` is the local variable name to assign to each item in `COLLECTION` , and `BODY` is what to render each iteration (which can reference `ITEM`). Here's an example:

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

output "for_directive" {
  value = "%{ for name in var.names }${name}, %{ endfor }"
}
```

When you run `terraform apply` , you get the following output:

```
$ terraform apply

(...)

Outputs:

for_directive = "neo, trinity, morpheus, "
```

There's also a version of the `for` string directive syntax that gives you the index in the for-loop:

```
%{ for <INDEX>, <ITEM> in <COLLECTION> }<BODY>%{ endfor }
```

Here's an example using the index:

```
output "for_directive_index" {
  value = "%{ for i, name in var.names }(${i}) ${name}, %{ endfor }"
}
```

When you run `terraform apply`, you get the following output:

```
$ terraform apply

(...)

Outputs:

for_directive_index = "(0) neo, (1) trinity, (2) morpheus, "
```

Note how in both outputs there is an extra trailing comma and space. You can fix this using conditionals—specifically, the `if` string directive—as described in the next section.

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

Used for conditional resources

`for_each` and `for` expressions

Used for conditional resources and inline blocks within a resource

`if` string directive

Used for conditionals within a string

Let's go through 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 begin by looking at if-statements in the next section and then move on to if-else-statements in the section thereafter.

If-statements with the count parameter

In Chapter 4, you created a Terraform module that could be used as a “blueprint” for deploying web server clusters. The module created an Auto Scaling Group (ASG), Application Load Balancer (ALB), security groups, and a number of other resources. One thing the module did *not* create was the scheduled action. Because you want to scale the cluster out only in production, you defined the `aws_autoscaling_schedule` resources directly in the production configurations under *live/prod/services/webserver-cluster/main.tf*. Is there a way you could define 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 you can use 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_during_business_hours" {
    scheduled_action_name  = "${var.cluster_name}-scale-out-during-business-
    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  = "${var.cluster_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:

- 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.
- 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_during_business_hours" {
  count = var.enable_autoscaling ? 1 : 0

  scheduled_action_name  = "${var.cluster_name}-scale-out-during-business-ho
  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  = "${var.cluster_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"
  db_remote_state_bucket = "(YOUR_BUCKET_NAME)"
  db_remote_state_key    = "stage/data-stores/mysql/terraform.tfstate"

  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` (make sure to also remove the custom `aws_autoscaling_schedule` resources that were in the production environment from Chapter 4):

```
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
  enable_autoscaling   = true

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

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 chapter, 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 allow the person applying the Terraform configurations to decide whether neo is assigned only read access or both read and write access. This is a slightly contrived example, but a useful one to demonstrate a simple type of if-else-statement.

Here is an IAM Policy that allows read-only access to CloudWatch:

```
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"`, 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_access" {
    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_only" {
    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_access" {
  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_only" {
  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). And there you are—you now know how to do if-else-statements!

Now that you have the ability to create one resource or the other based on an if/else condition, what do you do if you need to access an attribute on the resource that actually got created? For example, what if you wanted to add an output variable called `neo_cloudwatch_policy_arn`, which contains the ARN of the policy you actually attached?

The simplest option is to use ternary syntax:

```
output "neo_cloudwatch_policy_arn" {
  value = (
    var.give_neo_cloudwatch_full_access
    ? aws_iam_user_policy_attachment.neo_cloudwatch_full_access[0].policy_ar
    : aws_iam_user_policy_attachment.neo_cloudwatch_read_only[0].policy_arn
  )
}
```

This will work fine for now, but this code is a bit brittle: if you ever change the conditional in the `count` parameter of the `aws_iam_user_policy_attachment` resources—perhaps in the future, it'll depend on multiple variables and not solely on `var.give_neo_cloudwatch_full_access` —there's a risk that you'll forget to update the conditional in this output variable, and as a result, you'll get a very confusing error when trying to access an array element that might not exist.

A safer approach is to take advantage of the `concat` and `one` functions. The `concat` function takes two or more lists as inputs and combines them into a single list. The `one` function takes a list as input and if the list has 0 elements, it returns `null`; if the list has 1 element, it returns that element; and if the list has more than 1 element, it shows an error. Putting these two together, and combining them with a splat expression, you get the following:

```
output "neo_cloudwatch_policy_arn" {
  value = one(concat(
    aws_iam_user_policy_attachment.neo_cloudwatch_full_access[*].policy_arn,
    aws_iam_user_policy_attachment.neo_cloudwatch_read_only[*].policy_arn
  ))
}
```

Depending on the outcome of the if/else conditional, either `neo_cloudwatch_full_access` will be empty and `neo_cloudwatch_read_only` will contain one element or vice versa, so once you concatenate them together, you'll have a list with one element, and the `one` function will return that element. This will continue to work correctly no matter how you change your if/else conditional.

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, the result will be zero copies of the resource, inline block, or module where you have the `for_each`; if you pass it a nonempty collection, it will create one or more copies of the resource, inline block, or module. The only question is, how do you conditionally decide if 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 the `webserver-cluster` module in *modules/services/webserver-cluster/main.tf* sets 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, 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 uppercase (perhaps for consistency), and uses a conditional in the `for` expression to filter out any `key` set to `Name` because the module already sets its own `Name` tag. By filtering values in the `for` expression, you can implement arbitrary conditional logic.

Note that even though you should almost always prefer `for_each` over `count` for creating multiple copies of a resource or module, when it comes to conditional logic, setting `count` to 0 or 1 tends to be simpler than setting `for_each` to an empty or nonempty collection. Therefore, I typically recommend using `count` to conditionally create resources and modules, and using `for_each` for all other types of loops and conditionals.

Conditionals with the if String Directive

Let's now look at the `if` string directive, which has the following syntax:

```
%{ if <CONDITION> }<TRUEVAL>%{ endif }
```

where `CONDITION` is any expression that evaluates to a boolean and `TRUEVAL` is the expression to render if `CONDITION` evaluates to true.

Earlier in the chapter, you used the `for` string directive to do loops within a string to output several comma-separated names. The problem was that there was an extra trailing comma and space at the end of the string. You can use the `if` string directive to fix this issue as follows:

```
output "for_directive_index_if" {
  value = <<EOF
%{ for i, name in var.names }
  ${name}%{ if i < length(var.names) - 1 }, %{ endif }
%{ endfor }
EOF
}
```

There are a few changes here from the original version:

- I put the code in a *HEREDOC*, which is a way to define multiline strings. This allows me to spread the code out across several lines so it is more readable.
- I used the `if` string directive to not output the comma and space for the last item in the list.

When you run `terraform apply`, you get the following output:

```
$ terraform apply
```

```
(...)

Outputs:

for_directive_index_if = <<EOT

  neo,

  trinity,

  morpheus


EOT
```

Whoops. The trailing comma is gone, but we've introduced a bunch of extra whitespace (spaces and newlines). Every whitespace you put in a HEREDOC ends up in the final string. You can fix this by adding *strip markers* (~) to your string directives, which will eat up the extra whitespace before or after the strip marker:

```
output "for_directive_index_if_strip" {
  value = <<EOF
%{~ for i, name in var.names ~}
${name}%{ if i < length(var.names) - 1 }, %{ endif }
%{~ endfor ~}
EOF
}
```

Let's give this version a try:

```
$ terraform apply

(...)

Outputs:

for_directive_index_if_strip = "neo, trinity, morpheus"
```

OK, that's a nice improvement: no extra whitespace or commas. You can make this output even prettier by adding an `else` to the string directive, which uses the following syntax:

```
%{ if <CONDITION> }<TRUEVAL>%{ else }<FALSEVAL>%{ endif }
```

where `FALSEVAL` is the expression to render if `CONDITION` evaluates to false. Here's an example of how to use the `else` clause to add a period at the end:

```
output "for_directive_index_if_else_strip" {
  value = <<EOF
%{~ for i, name in var.names ~}
${name}%{ if i < length(var.names) - 1 }, %{ else }.%{ endif }
%{~ endfor ~}
EOF
}
```

When you run `terraform apply`, you get the following output:

```
$ terraform apply

(...)

Outputs:

for_directive_index_if_else_strip = "neo, trinity, morpheus."
```

Zero-Downtime Deployment

Now that your module has a clean and simple API for deploying a web server cluster, an important question to ask is, how do you update that cluster? That is, when you make changes to your code, how do you deploy a new Amazon Machine Image (AMI) across the cluster? And how do you do it without causing downtime for your users?

The first step is to expose the AMI as an input variable in *modules/services/webserver-cluster/variables.tf*. In real-world examples, this is all you would need because the actual web server code would be defined in the AMI. However, in the simplified examples in this book, all of the web server code is actually in the User Data script, and the AMI is just a vanilla Ubuntu image. Switching to a different version of Ubuntu won't make for much of a demonstration, so in addition to the new AMI input variable, you can also add an input variable to control the text the User Data script returns from its one-liner HTTP server:

```
variable "ami" {
  description = "The AMI to run in the cluster"
  type        = string
  default     = "ami-0fb653ca2d3203ac1"
}

variable "server_text" {
  description = "The text the web server should return"
  type        = string
  default     = "Hello, World"
}
```

Now you need to update the *modules/services/webserver-cluster/user-data.sh* Bash script to use this `server_text` variable in the `<h1>` tag it returns:

```
#!/bin/bash

cat > index.html <<EOF
<h1>${server_text}</h1>
<p>DB address: ${db_address}</p>
<p>DB port: ${db_port}</p>
EOF

nohup busybox httpd -f -p ${server_port} &
```

Finally, find the launch configuration in *modules/services/webserver-cluster/main.tf*, update the `image_id` parameter to use `var.ami`, and update the `templatefile` call in the `user_data` parameter to pass in `var.server_text`:

```
resource "aws_launch_configuration" "example" {
  image_id        = var.ami
  instance_type   = var.instance_type
  security_groups = [aws_security_group.instance.id]

  user_data       = templatefile("${path.module}/user-data.sh", {
    server_port = var.server_port
    db_address  = data.terraform_remote_state.db.outputs.address
    db_port     = data.terraform_remote_state.db.outputs.port
    server_text = var.server_text
  })

  # Required when using a launch configuration with an auto scaling group.
  lifecycle {
    create_before_destroy = true
  }
}
```

Now, in the staging environment, in *live/stage/services/webserver-cluster/main.tf*, you can set the new `ami` and `server_text` parameters:

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

  ami         = "ami-0fb653ca2d3203ac1"
```

```
    server_text = "New server text"

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

    instance_type      = "t2.micro"
    min_size           = 2
    max_size           = 2
    enable_autoscaling = false
  }
```

This code uses the same Ubuntu AMI, but changes the `server_text` to a new value. If you run the `plan` command, you should see something like the following:

```
  Terraform will perform the following actions:

    # module.webserver_cluster.aws_autoscaling_group.ex will be updated in-pla
    ~ resource "aws_autoscaling_group" "example" {
          id                          = "webservers-stage-terraform-20190516"
        ~ launch_configuration        = "terraform-20190516" -> (known after app
          (...)
      }

    # module.webserver_cluster.aws_launch_configuration.ex must be replaced
  +/- resource "aws_launch_configuration" "example" {
        ~ id                            = "terraform-20190516" -> (known after a
          image_id                      = "ami-0fb653ca2d3203ac1"
          instance_type                 = "t2.micro"
        ~ name                          = "terraform-20190516" -> (known after a
        ~ user_data                     = "bd7c0a6" -> "4919a13" # forces replac
          (...)
      }

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

As you can see, Terraform wants to make two changes: first, replace the old launch configuration with a new one that has the updated `user_data`; and second, modify the Auto Scaling Group in place to reference the new launch configuration. There is a problem here: merely referencing the new launch configuration will have no effect until the ASG launches new EC2 Instances. So how do you instruct the ASG to deploy new Instances?

One option is to destroy the ASG (e.g., by running `terraform destroy`) and then re-create it (e.g., by running `terraform apply`). The problem is that after you delete the old ASG, your users will experience downtime until the new ASG comes up. What you want to do instead is a *zero-down-*

time deployment. The way to accomplish that is to create the replacement ASG first and then destroy the original one. As it turns out, the `create_before_destroy` lifecycle setting you first saw in Chapter 2 does exactly this.

Here's how you can take advantage of this lifecycle setting to get a zero-downtime deployment:[1]

1. Configure the `name` parameter of the ASG to depend directly on the name of the launch configuration. Each time the launch configuration changes (which it will when you update the AMI or User Data), its name changes, and therefore the ASG's name will change, which forces Terraform to replace the ASG.
2. Set the `create_before_destroy` parameter of the ASG to `true` so that each time Terraform tries to replace it, it will create the replacement ASG before destroying the original.
3. Set the `min_elb_capacity` parameter of the ASG to the `min_size` of the cluster so that Terraform will wait for at least that many servers from the new ASG to pass health checks in the ALB before it will begin destroying the original ASG.

Here is what the updated `aws_autoscaling_group` resource should look like in *modules/services/webserver-cluster/main.tf*:

```
resource "aws_autoscaling_group" "example" {
  # Explicitly depend on the launch configuration's name so each time it's
  # replaced, this ASG is also replaced
  name = "${var.cluster_name}-${aws_launch_configuration.example.name}"

  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

  # Wait for at least this many instances to pass health checks before
  # considering the ASG deployment complete
  min_elb_capacity = var.min_size

  # When replacing this ASG, create the replacement first, and only delete t
  # original after
  lifecycle {
    create_before_destroy = true
  }

  tag {
    key                 = "Name"
```

```
      value                  = var.cluster_name
      propagate_at_launch = true
    }

    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
      }
    }
  }
```

If you rerun the `plan` command, you'll now see something that looks like the following:

```
Terraform will perform the following actions:

  # module.webserver_cluster.aws_autoscaling_group.example must be replaced
+/- resource "aws_autoscaling_group" "example" {
      ~ id      = "example-2019" -> (known after apply)
      ~ name    = "example-2019" -> (known after apply) # forces replacement
        (...)
    }

  # module.webserver_cluster.aws_launch_configuration.example must be replac
+/- resource "aws_launch_configuration" "example" {
      ~ id                  = "terraform-2019" -> (known after apply)
        image_id            = "ami-0fb653ca2d3203ac1"
        instance_type       = "t2.micro"
      ~ name                = "terraform-2019" -> (known after apply)
      ~ user_data           = "bd7c0a" -> "4919a" # forces replacement
        (...)
    }

    (...)

Plan: 2 to add, 2 to change, 2 to destroy.
```

The key thing to notice is that the `aws_autoscaling_group` resource now says `forces replacement` next to its name parameter, which means that Terraform will replace it with a new ASG running your new AMI or User Data. Run the `apply` command to kick off the deployment, and while it runs, consider how the process works.

You start with your original ASG running, say, v1 of your code (Figure 5-1).

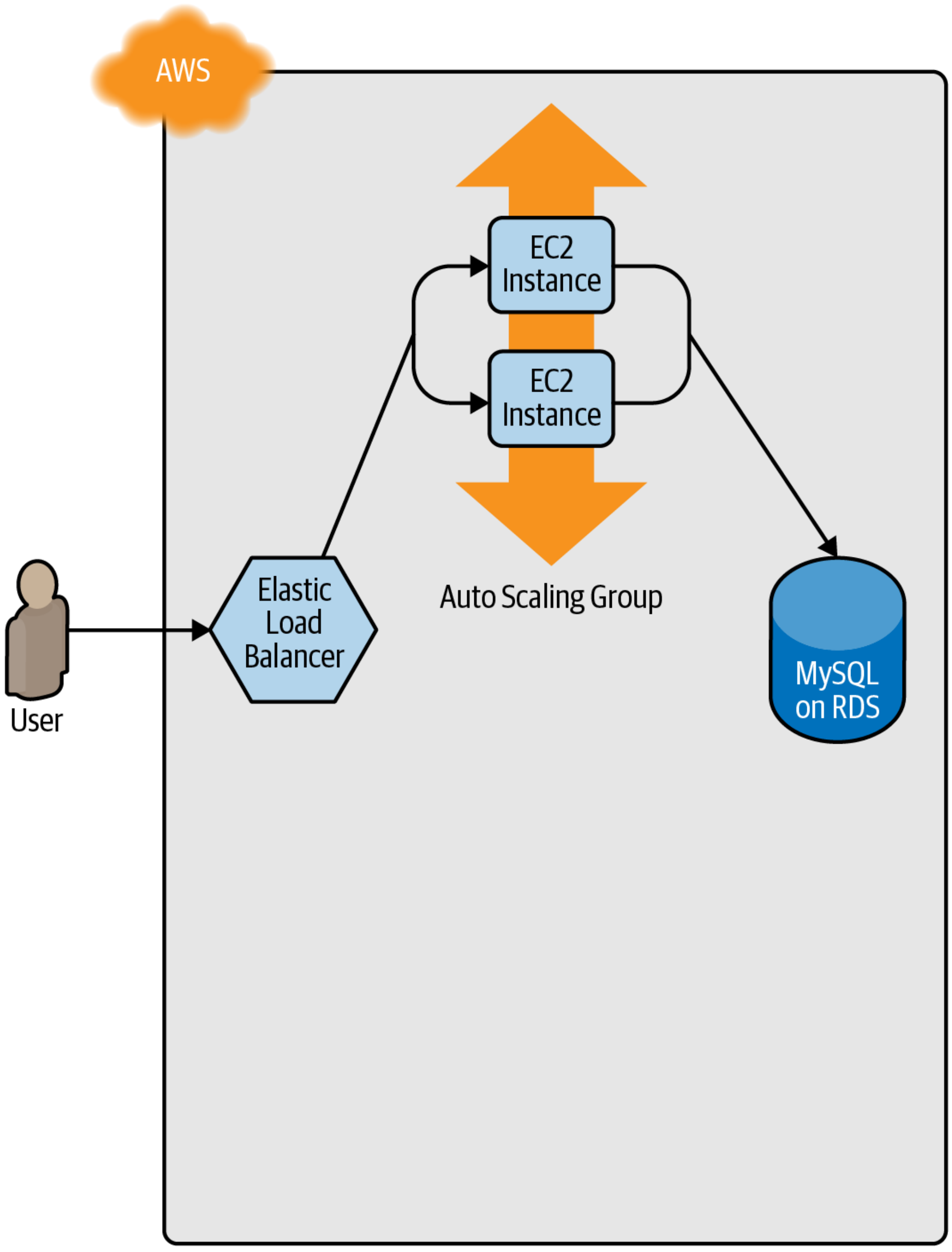


Figure 5-1. Initially, you have the original ASG running v1 of your code.

You make an update to some aspect of the launch configuration, such as switching to an AMI that contains v2 of your code, and run the `apply` command. This forces Terraform to begin deploying a new ASG with v2 of your code (Figure 5-2).

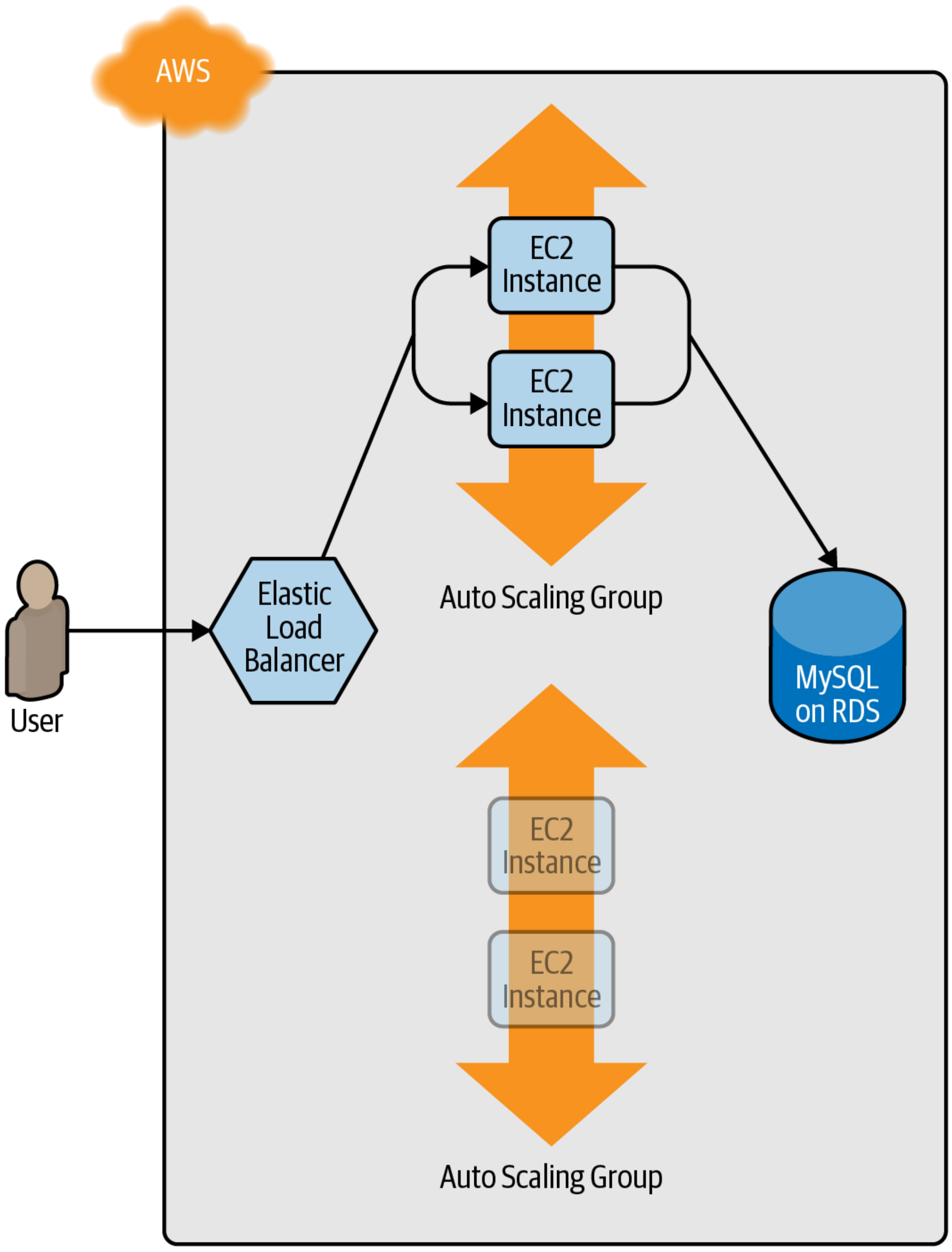


Figure 5-2. Terraform begins deploying the new ASG with v2 of your code.

After a minute or two, the servers in the new ASG have booted, connected to the database, registered in the ALB, and started to pass health checks. At this point, both the v1 and v2 versions of your app will be running simultaneously; and which one users see depends on where the ALB happens to route them (Figure 5-3).

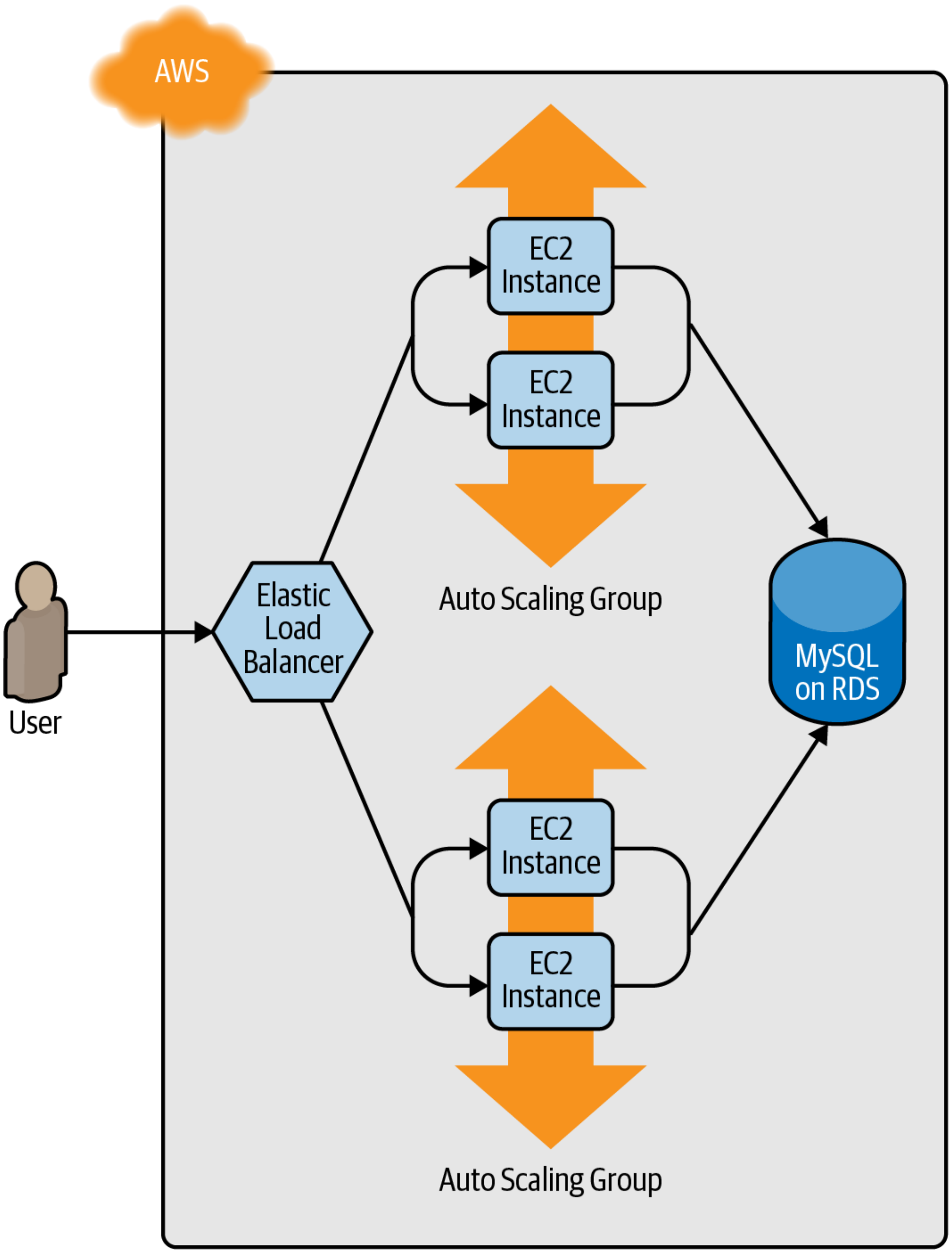


Figure 5-3. The servers in the new ASG boot up, connect to the DB, register in the ALB, and begin serving traffic.

After `min_elb_capacity` servers from the v2 ASG cluster have registered in the ALB, Terraform will begin to undeploy the old ASG, first by deregistering the servers in that ASG from the ALB, and then by shutting them down (Figure 5-4).

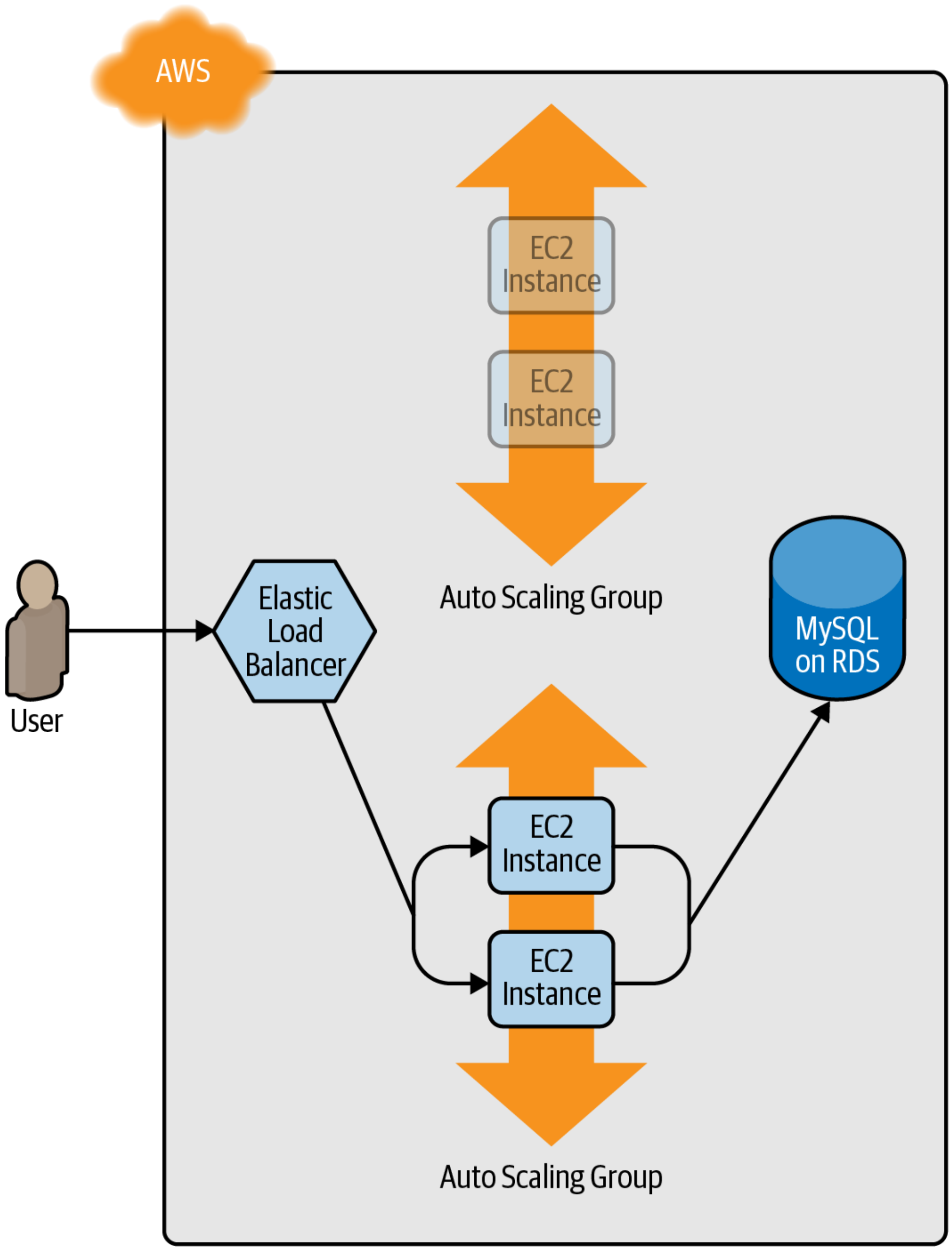


Figure 5-4. The servers in the old ASG begin to shut down.

After a minute or two, the old ASG will be gone, and you will be left with just v2 of your app running in the new ASG (Figure 5-5).

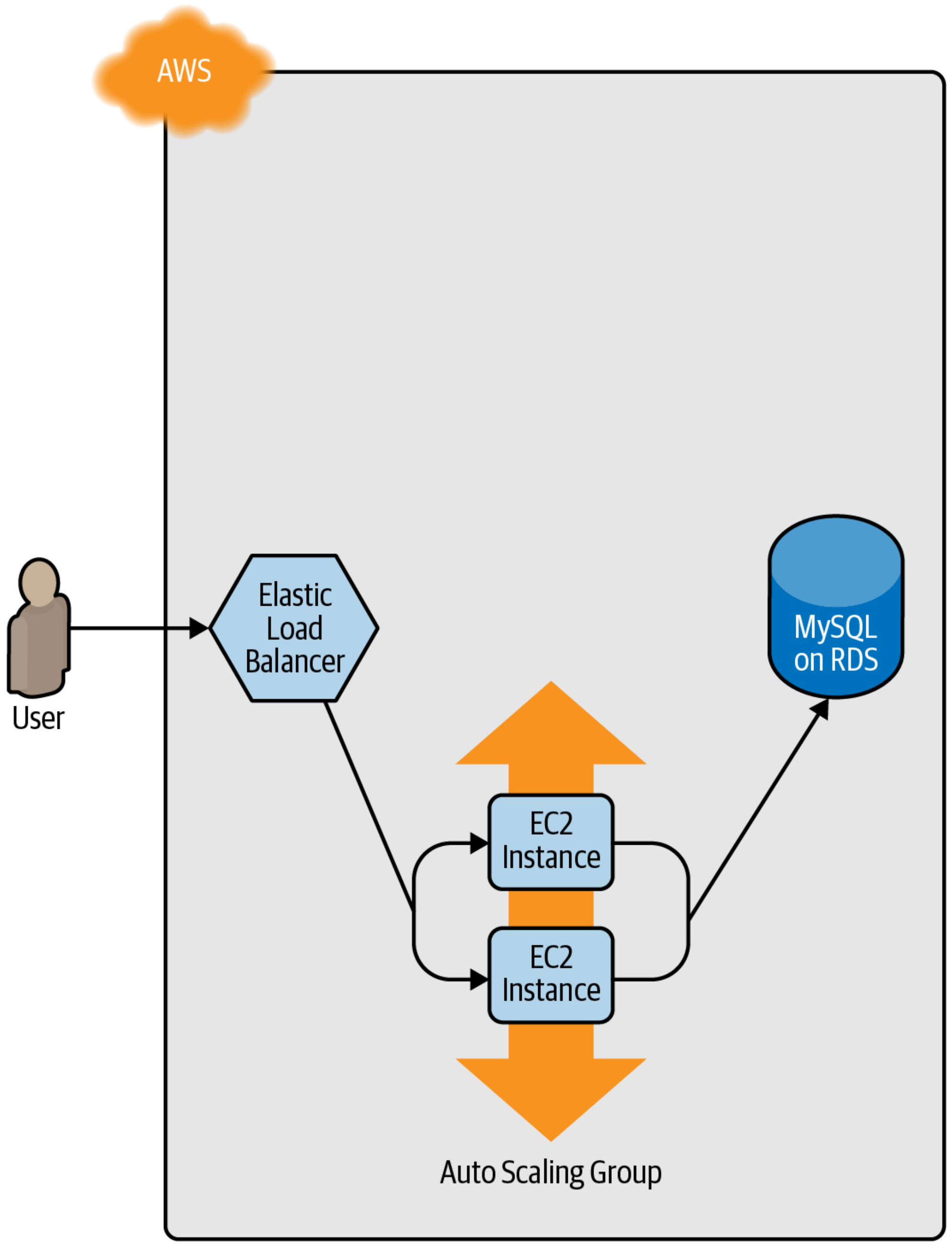
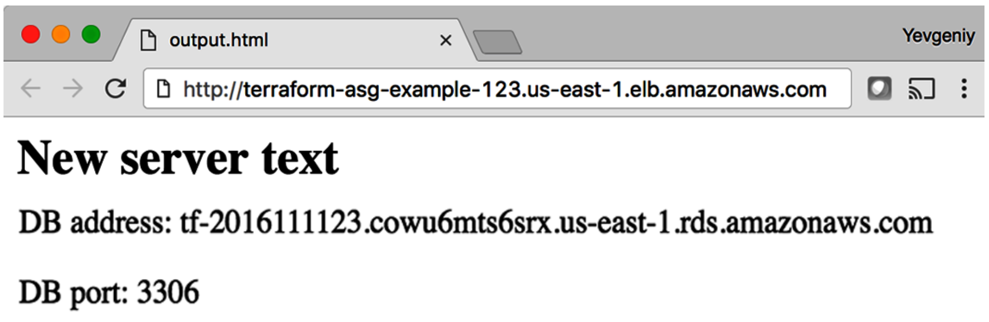


Figure 5-5. Now, only the new ASG remains, which is running v2 of your code.

During this entire process, there are always servers running and handling requests from the ALB, so there is no downtime. Open the ALB URL in your browser, and you should see something like Figure 5-6.


output.html
Yevgeniy
http://terraform-asg-example-123.us-east-1.elb.amazonaws.com

New server text

DB address: tf-2016111123.cowu6mts6srx.us-east-1.rds.amazonaws.com

DB port: 3306

Figure 5-6. The new code is now deployed.

Success! The new server text has deployed. As a fun experiment, make another change to the `server_text` parameter—for example, update it to say “foo bar”—and run the `apply` command. In a separate terminal tab, if you're on Linux/Unix/macOS, you can use a Bash one-liner to run `curl` in a loop, hitting your ALB once per second and allowing you to see the zero-downtime deployment in action:

```
$ while true; do curl http://<load_balancer_url>; sleep 1; done
```

For the first minute or so, you should see the same response: `New server text`. Then, you'll begin seeing it alternate between `New server text` and `foo bar`. This means the new Instances have registered in the ALB and passed health checks. After another minute, the `New server text` message will disappear, and you'll see only `foo bar`, which means the old ASG has been shut down. The output will look something like this (for clarity, I'm listing only the contents of the `<h1>` tags):

```
New server text
New server text
New server text
New server text
New server text
New server text
foo bar
New server text
foo bar
New server text
foo bar
New server text
foo bar
New server text
foo bar
New server text
foo bar
foo bar
foo bar
foo bar
foo bar
foo bar
```

As an added bonus, if something went wrong during the deployment, Terraform will automatically roll back. For example, if there were a bug in v2 of your app and it failed to boot, the Instances in the new ASG will not register with the ALB. Terraform will wait up to `wait_for_capacity_timeout` (default is 10 minutes) for `min_elb_capacity` servers of the v2 ASG to register in the ALB, after which it considers the deployment a failure, deletes the v2 ASG, and exits with an error (meanwhile, v1 of your app continues to run just fine in the original ASG).

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 the loop, if-statement, and deployment techniques, as well as those related to more general problems that affect Terraform as a whole:

- `count` and `for_each` have limitations.
- Zero-downtime deployment has limitations.
- Valid plans can fail.
- Refactoring can be tricky.

count and for_each Have Limitations

In the examples in this chapter, you made extensive use of the `count` parameter and `for_each` expressions in loops and if-statements. This works well, but there's an important limitation that you need to be aware of: you cannot reference any resource outputs in `count` or `for_each`.

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

```
resource "aws_instance" "example_1" {
  count         = 3
  ami           = "ami-0fb653ca2d3203ac1"
  instance_type = "t2.micro"
}
```

Because `count` is being set to a hardcoded value, this code will work without issues, and when you run `apply`, it will create three EC2 Instances. Now, what if you want 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-0fb653ca2d3203ac1"
  instance_type     = "t2.micro"
}

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

Again, this code works just fine, since `count` can reference data sources without problems. However, what happens if the number of instances you need to create depends 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-0fb653ca2d3203ac1"
  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. This means that `count` and `for_each` can reference hardcoded 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.

Zero-Downtime Deployment Has Limitations

There are a couple of gotchas with using `create_before_destroy` with an ASG to do zero-downtime deployment.

The first issue is that it doesn't work with auto scaling policies. Or, to be more accurate, it resets your ASG size back to its `min_size` after each deployment, which can be a problem if you had used auto scaling policies to increase the number of running servers. For example, the `webserver-cluster` module includes a couple of `aws_autoscaling_schedule` resources that increase the number of servers in the cluster from 2 to 10 at 9 a.m. If you ran a deployment at, say, 11 a.m., the replacement ASG would boot up with only 2 servers, rather than 10, and it would stay that way until 9 a.m. the next day. There are several possible workarounds, such as tweaking the `recurrence` parameter on the `aws_autoscaling_schedule` or setting the `desired_capacity` parameter of the ASG to get its value from a custom script that uses the AWS API to figure out how many instances were running before deployment.

However, the second, and bigger, issue is that, for important and complicated tasks like a zero-downtime deployment, you really want to use native, first-class solutions, and not workarounds that require you to haphazardly glue together `create_before_destroy`, `min_elb_capacity`, custom scripts, etc. As it turns out, for Auto Scaling Groups, AWS now offers a native solution called *instance refresh*.

Go back to your `aws_autoscaling_group` resource and undo the zero-downtime deployment changes:

- Set `name` back to `var.cluster_name`, instead of having it depend on the `aws_launch_configuration` name.
- Remove the `create_before_destroy` and `min_elb_capacity` settings.

And now, update the `aws_autoscaling_group` resource to instead use an `instance_refresh` block as follows:

```
resource "aws_autoscaling_group" "example" {
  name                 = var.cluster_name
  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

  # Use instance refresh to roll out changes to the ASG
  instance_refresh {
    strategy = "Rolling"
    preferences {
      min_healthy_percentage = 50
    }
    }
  }
```

If you deploy this ASG, and then later change some parameter (e.g., change `server_text`) and run `plan`, the diff will be back to just updating the `aws_launch_configuration`:

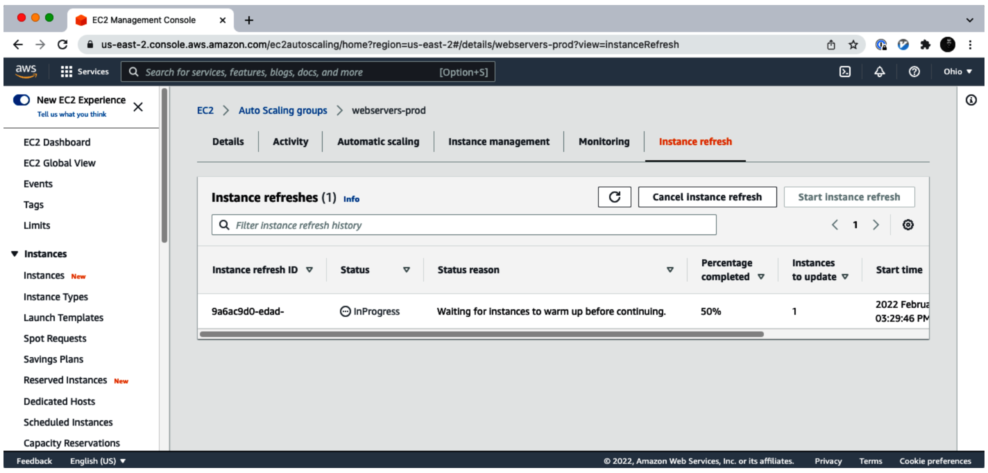
```
Terraform will perform the following actions:

  # module.webserver_cluster.aws_autoscaling_group.ex will be updated in-pla
  ~ resource "aws_autoscaling_group" "example" {
        id                          = "webservers-stage-terraform-20190516"
      ~ launch_configuration        = "terraform-20190516" -> (known after app
        (...)
    }

  # module.webserver_cluster.aws_launch_configuration.ex must be replaced
+/- resource "aws_launch_configuration" "example" {
      ~ id                              = "terraform-20190516" -> (known after a
        image_id                        = "ami-0fb653ca2d3203ac1"
        instance_type                   = "t2.micro"
      ~ name                            = "terraform-20190516" -> (known after a
      ~ user_data                       = "bd7c0a6" -> "4919a13" # forces replac
        (...)
    }

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

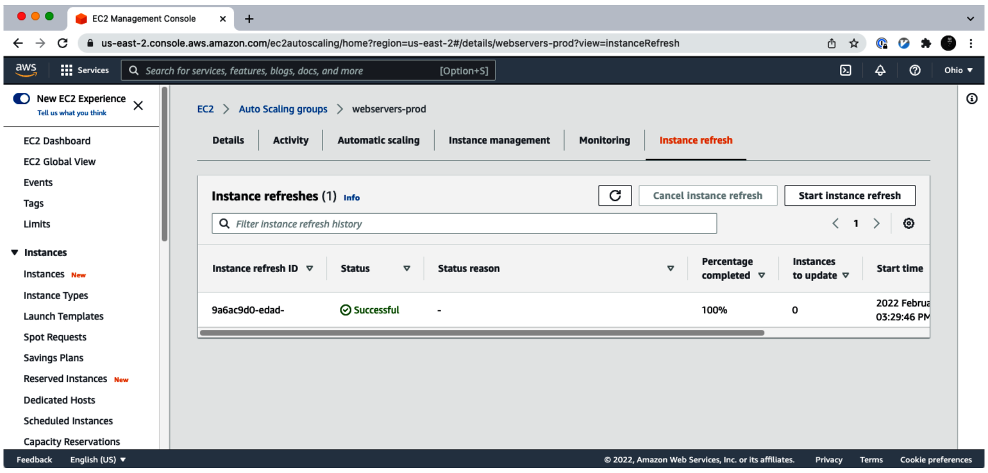
If you run `apply`, it'll complete very quickly, and at first, nothing new will be deployed. However, in the background, because you modified the launch configuration, AWS will kick off the instance refresh process, as shown in Figure 5-7.



Instance refresh ID	Status	Status reason	Percentage completed	Instances to update	Start time
9a6ac9d0-edad-	InProgress	Waiting for instances to warm up before continuing.	50%	1	2022 Februa 03:29:46 PM

Figure 5-7. An instance refresh is in progress.

AWS will initially launch one new instance, wait for it to pass health checks, shut down one of the older instances, and then repeat the process with the second instance, until the instance refresh is completed, as shown in Figure 5-8.



Instance refresh ID	Status	Status reason	Percentage completed	Instances to update	Start time
9a6ac9d0-edad-	✓ Successful	-	100%	0	2022 Februa 03:29:46 PM

Figure 5-8. An instance refresh is completed.

This process is entirely managed by AWS, is reasonably configurable, handles errors pretty well, and requires no workarounds. The only drawback is the process can sometimes be slow (taking up to 20 minutes to replace just two servers), but other than that, it's a much more robust solution to use for most zero-downtime deployments.

In general, you should prefer to use first-class, native deployment options like instance refresh whenever possible. Although such options weren't always available in the earlier days of Terraform, these days, many resources support native deployment options. For example, if you're using Amazon Elastic Container Service (ECS) to deploy Docker containers, the `aws_ecs_service` resource natively supports zero-downtime deployments via the `deployment_maximum_percent` and `deployment_minimum_healthy_percent` parameters; if you're using Kubernetes to deploy Docker containers, the `kubernetes_deployment` resource natively supports zero-downtime deployments by setting the `strategy` parameter to `RollingUpdate` and providing configuration via the `rolling_update` block. Check the docs for the resources you're using, and make use of native functionality when you can!

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 manually in Chapter 2:

```
resource "aws_iam_user" "existing_user" {
  # Make sure to update this to your own user name!
  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.

  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 just with IAM users but with 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` looks only 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 will still fail.

There are two main lessons to take away from this:

After you start using Terraform, you should only use Terraform.

> When a part of your infrastructure is managed by Terraform, you should never manually make changes 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, given that the code will no longer be an accurate representation of your infrastructure.

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 that Terraform is aware of and can manage that infrastructure. The `import` command takes two arguments. The first argument is the "address" of the resource in your Terraform configuration files. This makes use of the same syntax as resource references, such as `<PROVIDER>_<TYPE>.<NAME>` (e.g., `aws_iam_user.existing_user`). The second argument is a resource-specific ID that identifies the resource to import. For example, the ID for an `aws_iam_user` resource is the name of the user (e.g., yevgeniy.brikman), and the ID for an `aws_instance` is the EC2 Instance ID (e.g., i-190e22e5). The documentation at the bottom of the page for each resource typically specifies how to import it.

For example, here is the `import` command that you can use to sync the `aws_iam_user` you just added in your Terraform configurations with the IAM user you created back in Chapter 2 (obviously, you should replace "yevgeniy.brikman" with your own username in this command):

```
$ terraform import aws_iam_user.existing_user yevgeniy.brikman
```

Terraform will use the AWS API to find your IAM user and create an association in its state file between that user and the `aws_iam_user.existing_user` resource in your Terraform configurations. From then on, when you run the `plan` command, Terraform will know that an IAM user already exists and not try to create it again.

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 might want to look into tools such as terraformer and terracognita, which can import both code and state from supported cloud environments automatically.

Refactoring Can Be Tricky

A common programming practice is *refactoring*, in which 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 IaC 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 automatically rename the variable or function for you, across the entire codebase. Although such a renaming is something you might do without thinking twice in a general-purpose programming language, you need to be very careful about how you do it in Terraform, or it could lead to an outage.

For example, the `webserver-cluster` module 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 that you want to rename the service to `bar`. This might seem like a trivial change, but it can actually cause an outage!

That's because the `webserver-cluster` module uses the `cluster_name` variable in a number of resources, including the `name` parameters of two security groups and the ALB:

```
resource "aws_lb" "example" {
  name               = var.cluster_name
  load_balancer_type = "application"
  subnets            = data.aws_subnets.default.ids
  security_groups    = [aws_security_group.alb.id]
}
```

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 an ALB, there will be nothing to route traffic to your web server cluster until the new ALB 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 that you might 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`, 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. You may run into similar problems if you change the identifier associated with a module, split one module into multiple modules, or add `count` or `for_each` to a resource or module that didn't have it before.

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

Always use the `plan` command

You can catch all of these gotchas 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.

Create before destroy

If you do want to replace a resource, think carefully about whether its replacement should be created before you delete the original. If so, you might be able to use `create_before_destroy` to make that happen. 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.

Refactoring may require changing state

If you want to refactor your code without accidentally causing downtime, you'll need to update the Terraform state accordingly. However, you should never update Terraform state files by hand! Instead, you have two options: do it manually by running `terraform state mv` commands, or do it automatically by adding a `moved` block to your code.

Let's first look at the `terraform state mv` command, which has the following syntax:

```
terraform state mv <ORIGINAL_REFERENCE> <NEW_REFERENCE>
```

where `ORIGINAL_REFERENCE` is the reference expression to the resource as it is now and `NEW_REFERENCE` is the new location you want to move it to. For example, if you're renaming an `aws_security_group` group from `instance` to `cluster_instance`, you could run the following:

```
$ terraform state mv \
  aws_security_group.instance \
  aws_security_group.cluster_instance
```

This instructs Terraform that the state that used to be associated with `aws_security_group.instance` should now be associated with `aws_security_group.cluster_instance`. If you rename an identifier and run this command, you'll know you did it right if the subsequent `terraform plan` shows no changes.

Having to remember to run CLI commands manually is error prone, especially if you refactored a module used by dozens of teams in your company, and each of those teams needs to remember to run `terraform state mv` to avoid downtime. Fortunately, Terraform 1.1 has added a way to handle this automatically: `moved` blocks. Any time you refactor your code, you should add a `moved` block to capture how the state should be updated. For example, to capture that the `aws_security_group` resource was renamed from `instance` to `cluster_instance`, you would add the following `moved` block:

```
moved {
  from = aws_security_group.instance
  to   = aws_security_group.cluster_instance
}
```

Now, whenever anyone runs `apply` on this code, Terraform will automatically detect if it needs to update the state file:

```
Terraform will perform the following actions:

  # aws_security_group.instance has moved to
  # aws_security_group.cluster_instance
    resource "aws_security_group" "cluster_instance" {
        name                   = "moved-example-security-group"
        tags                   = {}
        # (8 unchanged attributes hidden)
    }

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

Do you want to perform these actions?
  Terraform will perform the actions described above.
  Only 'yes' will be accepted to approve.

  Enter a value:
```

If you enter **`yes`**, Terraform will update the state automatically, and as the plan shows no resources to add, change, or destroy, Terraform will make no other changes—which is exactly what you want!

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 get used to checking the documentation. And, once again, make sure to always use the `plan` command and consider whether you should use a `create_before_destroy` strategy.

Conclusion

Although Terraform is a declarative language, it includes a large number of tools, such as variables and modules, which you saw in Chapter 4, and `count`, `for_each`, `for`, `create_before_destroy`, and built-in functions, which you saw in this chapter, that give the language a surprising amount of flexibility and expressive power. There are many permutations of the if-statement tricks shown in this chapter, so spend some time browsing the functions documentation, and let your inner hacker go wild. OK, maybe not too wild, as someone still needs to maintain your code, but just wild enough that you can create clean, beautiful APIs for your modules.

Let's now move on to Chapter 6, where I'll go over how create modules that are not only clean and beautiful but also handle secrets and sensitive data in a safe and secure manner.

1 Credit for this technique goes to Paul Hinze.