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

Terraform is a declarative language. As discussed in Chapter 1, infrastructure as code 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, since 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, a meta-parameter called `count`, a lifecycle block called `create_before_destroy`, a ternary operator, plus a large number of interpolation functions—that allow you to do certain types of loops, if-statements, and zero-downtime deployments. You probably won't need to use these too often, but when you do, it's good to be aware of what's possible and what the gotchas are. Here are the topics I'll cover in this chapter:

- Loops
- If-statements
- If-else-statements
- Zero-downtime deployment
- Terraform gotchas

EXAMPLE CODE

As a reminder, all of the code examples in the book can be found at the following URL: *https://github.com/brikis98/terraform-up-and-running-code*.

Loops

In Chapter 2, you 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, which should live in *live/global/iam/main.tf*:

```
provider "aws" {
  region = "us-east-1"
}

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

This code uses the `aws_iam_user` resource to create a single new IAM user

```
# 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, almost every Terraform resource has a meta-parameter you can use called `count`. This parameter defines 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 different name ("neo.0", "neo.1", "neo.2"):

```
+ aws_iam_user.example.0
    arn:           "<computed>"
    force_destroy: "false"
    name:          "neo.0"
    path:          "/"
    unique_id:     "<computed>"

+ aws_iam_user.example.1
    arn:           "<computed>"
    force_destroy: "false"
    name:          "neo.1"
    path:          "/"
    unique_id:     "<computed>"

+ aws_iam_user.example.2
    arn:           "<computed>"
    force_destroy: "false"
    name:          "neo.2"
    path:          "/"
    unique_id:     "<computed>"


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 interpolation functions built into 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/vars.tf*:

```
variable "user_names" {
  description = "Create IAM users with these names"
  type        = "list"
  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` and two interpolation functions, `element` and `length`:

```
"${element(LIST, INDEX)}"
"${length(LIST)}"
```

The `element` function returns the item located at `INDEX` in the given `LIST`. [1] The `length` function returns the number of items in `LIST` (it also works with strings and maps). Putting these together, you get:

```
resource "aws_iam_user" "example" {
  count = "${length(var.user_names)}"
  name  = "${element(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 name:

```
+ aws_iam_user.example.0
    arn:           "<computed>"
    force_destroy: "false"
    name:          "neo"
    path:          "/"
    unique_id:     "<computed>"

+ aws_iam_user.example.1
    arn:           "<computed>"
    force_destroy: "false"
    name:          "trinity"
    path:          "/"
    unique_id:     "<computed>"

+ aws_iam_user.example.2
    arn:           "<computed>"
    force_destroy: "false"
    name:          "morpheus"
    path:          "/"
    unique_id:     "<computed>"


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 (`TYPE.NAME.ATTRIBUTE`), you have to specify which IAM user you're interested in by specifying its index in the list:

```
"${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}"
}
```

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

```
"${aws_iam_user.example.*.arn}"
```

When you use the splat character, you get back a list, so you need to wrap the output variable with brackets:

```
output "all_arns" {
  value = ["${aws_iam_user.example.*.arn}"]
}
```

When you run the `apply` command, the `all_arns` output will contain the list of 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
]
```

Note that since the splat syntax returns a list, you can combine it with other interpolation functions, such as `element`. For example, let's say you wanted to give each of these IAM users read-only access to EC2. You may remember from Chapter 2 that by default, new IAM users have no permissions whatsoever, and that to grant permissions, you can attach IAM policies to those IAM users. An IAM policy is a JSON document:

```
{
  "Statement": [
    {
      "Effect": "Allow",
      "Action": ["ec2:Describe*"],
      "Resource": ["*"]
    }
  ]
}
```

An IAM policy consists of one or more *statements*, each of which specifies an *effect* (either "Allow" or "Deny"), on one or more *actions* (e.g., `"ec2:Describe*"` allows all API calls to EC2 that start with the name `"Describe"`), on one or more *resources* (e.g., `"*"` means "all resources"). Although you can define IAM policies in JSON, Terraform also provides a handy data source called the `aws_iam_policy_document` that gives you a more concise way to define the same IAM policy:

```
data "aws_iam_policy_document" "ec2_read_only" {
  statement {
    effect    = "Allow"
    actions   = ["ec2:Describe*"]
    resources = ["*"]
  }
}
```

To create a new managed IAM policy from this document, you need to use the `aws_iam_policy` resource and set its `policy` parameter to the JSON output of the `aws_iam_policy_document` you just created:

```
resource "aws_iam_policy" "ec2_read_only" {
  name   = "ec2-read-only"
  policy = "${data.aws_iam_policy_document.ec2_read_only.json}"
}
```

Finally, to attach the IAM policy to your new IAM users, you use the `aws_iam_user_policy_attachment` resource:

```
resource "aws_iam_user_policy_attachment" "ec2_access" {
  count      = "${length(var.user_names)}"
  user       = "${element(aws_iam_user.example.*.name, count.index)}"
  policy_arn = "${aws_iam_policy.ec2_read_only.arn}"
}
```

This code uses the `count` parameter to "loop" over each of your IAM users and the `element` interpolation function to select each user's ARN from the list returned by `aws_iam_user.example.*.arn`.

If-Statements

Using `count` lets you do a basic loop. If you're clever, you can use the same mechanism to do a basic if-statement as well. Let's start by looking at simple if-statements in the next section and then move on to more complicated ones in the section after that.

Simple If-Statements

In Chapter 4, you created a Terraform module that could be used as "blueprint" for deploying web server clusters. The module created an Auto Scaling Group (ASG), Elastic Load Balancer (ELB), 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, 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/vars.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"
}
```

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 ${var.enable_autoscaling} {
  resource "aws_autoscaling_schedule" "scale_out_during_business_hours" {
    scheduled_action_name  = "scale-out-during-business-hours"
    min_size               = 2
    max_size               = 10
    desired_capacity       = 10
    recurrence             = "0 9 * * *"
    autoscaling_group_name = "${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. In Terraform, if you set a variable to a boolean `true` (that is, the word `true` without any quotes around it), it will be coerced into a 1, and if you set it to a boolean `false`, it will be coerced into a 0.
2. 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.

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}"

  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}"

  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"
  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
}
```

More Complicated If-Statements

This approach works well if the user passes an explicit boolean value to your module, but what do you do if the boolean is the result of a more complicated comparison, such as string equality? To handle more complicated cases, you can again use the `count` parameter, but this time, rather than setting it to a boolean variable, you set it to the value returned by a *conditional*. Conditionals in Terraform use the same *ternary syntax* available in many programming languages:

```
"${CONDITION ? TRUEVAL : FALSEVAL}"
```

For example, a more verbose way to do the simple if-statement from the previous section is as follows:

```
count = "${var.enable_autoscaling ? 1 : 0}"
```

Let's go through a more complicated example. Imagine that as part of the `webserver-cluster` module, you wanted to create a set of CloudWatch alarms. A *CloudWatch alarm* can be configured to notify you via a variety of mechanisms (e.g., email, text message) if a specific metric exceeds a predefined threshold. For example, here is how you could use the `aws_cloudwatch_metric_alarm` resource in *modules/services/webserver-cluster/main.tf* to create an alarm that goes off if the average CPU utilization in the cluster is over 90% during a 5-minute period:

```
resource "aws_cloudwatch_metric_alarm" "high_cpu_utilization" {
  alarm_name  = "${var.cluster_name}-high-cpu-utilization"
  namespace   = "AWS/EC2"
  metric_name = "CPUUtilization"

  dimensions = {
    AutoScalingGroupName = "${aws_autoscaling_group.example.name}"
  }

  comparison_operator = "GreaterThanThreshold"
  evaluation_periods  = 1
  period              = 300
  statistic           = "Average"
  threshold           = 90
  unit                = "Percent"
}
```

This works fine for a CPU Utilization alarm, but what if you wanted to add another alarm that goes off when CPU credits are low? [2] Here is a CloudWatch alarm that goes off if your web server cluster is almost out of CPU credits:

```
resource "aws_cloudwatch_metric_alarm" "low_cpu_credit_balance" {
  alarm_name = "${var.cluster_name}-low-cpu-credit-balance"
  namespace   = "AWS/EC2"
  metric_name = "CPUCreditBalance"

  dimensions = {
    AutoScalingGroupName = "${aws_autoscaling_group.example.name}"
  }

  comparison_operator = "LessThanThreshold"
  evaluation_periods  = 1
  period              = 300
  statistic           = "Minimum"
  threshold           = 10
  unit                = "Count"
}
```

The catch is that CPU credits only apply to tXXX Instances (e.g., `t2.micro`, `t2.medium`, etc). Larger instance types (e.g., `m4.large`) don't use CPU credits and don't report a `CPUCreditBalance` metric, so if you create such an alarm for those instances, the alarm will always be stuck in the "INSUFFICIENT_DATA" state. Is there a way to create an alarm only if `var.instance_type` starts with the letter "t"?

You could add a new boolean input variable called `var.is_t2_instance`, but that would be redundant with `var.instance_type`, and you'd most likely forget to update one when updating the other. A better alternative is to use a conditional:

```
resource "aws_cloudwatch_metric_alarm" "low_cpu_credit_balance" {
  count = "${format("%.1s", var.instance_type) == "t" ? 1 : 0}"

  alarm_name = "${var.cluster_name}-low-cpu-credit-balance"
  namespace   = "AWS/EC2"
  metric_name = "CPUCreditBalance"

  dimensions = {
    AutoScalingGroupName = "${aws_autoscaling_group.example.name}"
  }

  comparison_operator = "LessThanThreshold"
  evaluation_periods  = 1
  period              = 300
  statistic           = "Minimum"
  threshold           = 10
  unit                = "Count"
}
```

The alarm code is the same as before, except for the relatively complicated `count` parameter:

```
count = "${format("%.1s", var.instance_type) == "t" ? 1 : 0}"
```

This code uses the `format` function to extract just the first character from `var.instance_type`. If that character is a "t" (e.g., `t2.micro`), it sets the `count` to 1; otherwise, it sets the count to 0. This way, the alarm is only created for instance types that actually have a `CPUCreditBalance` metric.

If-Else-Statements

Now that you know how to do an if-statement, what about an if-else-statement? Let's again start by looking at simple if-else-statements in the next section and move on to more complicated ones in the section after that.

Simple If-Else-Statements

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 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, where all that matters is that one of the if or else branches gets executed, and the rest of the Terraform code doesn't need to know which one.

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
    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"
}
```

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 again use the `count` parameter and a boolean, but this time, you also need to take advantage of the fact that Terraform allows simple math in interpolations:

```
resource "aws_iam_user_policy_attachment" "neo_cloudwatch_full_access"
  count = "${var.give_neo_cloudwatch_full_access}"

  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 = "${1 - var.give_neo_cloudwatch_full_access}"

  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, sets its `count` parameter to `var.give_neo_cloudwatch_full_access`, so this resource only gets created if `var.give_neo_cloudwatch_full_access` is `true` (this

is the if-clause). The second one, which attaches the CloudWatch read-only permissions, sets its `count` parameter to `1 - var.give_neo_cloudwatch_full_access`, so it will have the inverse behavior, and only be created if `var.give_neo_cloudwatch_full_access` is `false` (this is the else-clause).

More Complicated If-Else-Statements

This approach works well if your Terraform code doesn't need to know which of the if or else clauses actually got executed. But what if you need to access some output attribute on the resource that comes out of the if or else clause? For example, what if you wanted to offer two different User Data scripts in the `webserver-cluster` module and allow users to pick which one gets executed? Currently, the `webserver-cluster` module pulls in the *user-data.sh* script via a `template_file` data source:

```
data "template_file" "user_data" {
  template = "${file("${path.module}/user-data.sh")}"

  vars {
    server_port = "${var.server_port}"
    db_address  = "${data.terraform_remote_state.db.address}"
    db_port     = "${data.terraform_remote_state.db.port}"
  }
}
```

The current *user-data.sh* script looks like this:

```
#!/bin/bash

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

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

Now, imagine that you wanted to allow some of your web server clusters to use this alternative, shorter script, called *user-data-new.sh*:

```
#!/bin/bash

echo "Hello, World, v2" > index.html
nohup busybox httpd -f -p "${server_port}" &
```

To use this script, you need a new `template_file` data source:

```
data "template_file" "user_data_new" {
  template = "${file("${path.module}/user-data-new.sh")}"

  vars {
    server_port = "${var.server_port}"
  }
}
```

The question is, how can you allow the user of the `webserver-cluster` module to pick from one of these User Data scripts? As a first step, you could add a new boolean input variable in *modules/services/webserver-cluster/vars.tf*:

```
variable "enable_new_user_data" {
  description = "If set to true, use the new User Data script"
}
```

If you were using a general-purpose programming language, you could add an if-else-statement to the launch configuration to pick between the two User Data `template_file` options as follows:

```
# This is just pseudo code. It won't actually work in Terraform.
resource "aws_launch_configuration" "example" {
  image_id        = "ami-40d28157"
  instance_type   = "${var.instance_type}"
  security_groups = ["${aws_security_group.instance.id}"]

  if ${var.enable_new_user_data} {
    user_data = "${data.template_file.user_data_new.rendered}"
  } else {
    user_data = "${data.template_file.user_data.rendered}"
  }

  lifecycle {
    create_before_destroy = true
  }
}
```

To make this work with real Terraform code, you first need to use the if-else-statement trick from before to ensure that only one of the `template_file` data sources is actually created:

```
data "template_file" "user_data" {
  count = "${1 - var.enable_new_user_data}"

  template = "${file("${path.module}/user-data.sh")}"

  vars {
```

```
    server_port = "${var.server_port}"
    db_address  = "${data.terraform_remote_state.db.address}"
    db_port     = "${data.terraform_remote_state.db.port}"
  }
}

data "template_file" "user_data_new" {
  count = "${var.enable_new_user_data}"

  template = "${file("${path.module}/user-data-new.sh")}"

  vars {
    server_port = "${var.server_port}"
  }
}
```

If `var.enable_new_user_data` is `true`, then `data.template_file.user_data_new` will be created and `data.template_file.user_data` will not; if it's `false`, it'll be the other way around. All you have to do now is to set the `user_data` parameter of the `aws_launch_configuration` resource to the `template_file` that actually exists. To do this, you can take advantage of the `concat` interpolation function:

```
"${concat(LIST1, LIST2, ...)}"
```

The `concat` function combines two or more lists into a single list. Here is how you can combine it with the `element` function to select the proper `template_file`:

```
resource "aws_launch_configuration" "example" {
  image_id        = "ami-40d28157"
  instance_type   = "${var.instance_type}"
  security_groups = ["${aws_security_group.instance.id}"]

  user_data = "${element(
    concat(data.template_file.user_data.*.rendered,
           data.template_file.user_data_new.*.rendered),
    0)}"

  lifecycle {
    create_before_destroy = true
  }
}
```

Let's break the large value for the `user_data` parameter down. First, take a look at the inner part:

```
concat(data.template_file.user_data.*.rendered,
       data.template_file.user_data_new.*.rendered)
```

Note that the two `template_file` resources are both lists, as they both use the `count` parameter. One of these lists will be of length 1 and the other of length 0, depending on the value of `var.enable_new_user_data`. The preceding code uses the `concat` function to combine these two lists into a single list, which will be of length 1. Now consider the outer part:

```
user_data =  "${element(<INNER>, 0)}"
```

This code simply takes the list returned by the inner part, which will be of length 1, and uses the `element` function to extract that one value.

You can now try out the new User Data script in the staging environment by setting the `enable_new_user_data` parameter to `true` in *live/stage/services/webserver-cluster/main.tf*:

```
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
  enable_new_user_data = true
}
```

In the production environment, you can stick with the old version of the script by setting `enable_new_user_data` to `false` in *live/prod/services/webserver-cluster/main.tf*:

```
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
  enable_new_user_data = false
}
```

Using `count` and interpolation 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.

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 have made changes to your code, how do you deploy a new 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/vars.tf*. In real-world examples, this is all you would need, as 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"
  default     = "ami-40d28157"
}

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

Earlier in the chapter, to practice with if-else-statements, you created two User Data scripts. Let's consolidate that back down to one to keep things simple. First, in *modules/services/webserver-cluster/vars.tf*, remove the `enable_new_user_data` input variable. Second, in *modules/services/webserver-cluster/main.tf*, remove the `template_file` resource called `user_data_new`. Third, in the same file, update the other `template_file` resource, called `user_data`, to no longer use the `enable_new_user_data` input variable, and to add the new `server_text` input variable to its `vars` block:

```
data "template_file" "user_data" {
  template = "${file("${path.module}/user-data.sh")}"

  vars {
    server_port = "${var.server_port}"
    db_address  = "${data.terraform_remote_state.db.address}"
    db_port     = "${data.terraform_remote_state.db.port}"
    server_text = "${var.server_text}"
  }
}
```

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*, set its `user_data` parameter to the remaining `template_file` (the one called `user_data`), and set its `ami` parameter to the new `ami` input variable:

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

  user_data = "${data.template_file.user_data.rendered}"

  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 and remove the `enable_new_user_data` parameter:

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

  ami         = "ami-40d28157"
```

```
  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 (I've omitted some of the output for clarity):

```
~ module.webserver_cluster.aws_autoscaling_group.example
    launch_configuration:
      "terraform-2016182624wu" => "${aws_launch_configuration.example.i

-/+ module.webserver_cluster.aws_launch_configuration.example
    ebs_block_device.#:  "0" => "<computed>"
    ebs_optimized:       "false" => "<computed>"
    enable_monitoring:   "true" => "true"
    image_id:            "ami-40d28157" => "ami-40d28157"
    instance_type:       "t2.micro" => "t2.micro"
    key_name:            "" => "<computed>"
    name:                "terraform-2016wu" => "<computed>"
    root_block_device.#: "0" => "<computed>"
    security_groups.#:   "1" => "1"
    user_data:           "416115339b" => "3bab6ede8dc" (forces new reso


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 to reference the new launch configuration. The problem is that merely referencing the new launch configuration will have no effect until the Auto Scaling Group launches new EC2 Instances. So how do you tell the Auto Scaling Group 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-downtime deployment*. The way to accomplish that is to create the replacement ASG first and then destroy the original one. As it turns out, this is exactly what the `create_before_destroy` lifecycle setting does!

Here's how you can take advantage of this lifecycle setting to get a zero-downtime deployment: [3]

1. Configure the `name` parameter of the ASG to depend directly on the name of the launch configuration. That way, each time the launch configuration changes (which it will when you update the AMI or User Data), Terraform will try to replace the ASG.
2. Set the `create_before_destroy` parameter of the ASG to `true`, so each time Terraform tries to replace it, it will create the replacement 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 register in the ELB before it'll start 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" {
  name = "${var.cluster_name}-${aws_launch_configuration.example.name}"

  launch_configuration = "${aws_launch_configuration.example.id}"
  availability_zones   = ["${data.aws_availability_zones.all.names}"]
  load_balancers       = ["${aws_elb.example.name}"]
  health_check_type    = "ELB"

  min_size         = "${var.min_size}"
  max_size         = "${var.max_size}"
  min_elb_capacity = "${var.min_size}"

  lifecycle {
    create_before_destroy = true
  }

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

As you may remember, a gotcha with the `create_before_destroy` parameter is that if you set it to `true` on a resource R, you also have to set it to `true` on every resource that R depends on. In the web server cluster module, the

`aws_autoscaling_group` resource depends on one other resource, the `aws_elb`. The `aws_elb`, in turn, depends on one other resource, an `aws_security_group`. Set `create_before_destroy` to `true` on both of those resources.

If you rerun the `plan` command, you'll now see something that looks like this (I've omitted some of the output for clarity):

```
-/+ module.webserver_cluster.aws_autoscaling_group.example
    availability_zones.#:      "4" => "4"
    default_cooldown:          "300" => "<computed>"
    desired_capacity:          "2" => "<computed>"
    force_delete:              "false" => "false"
    health_check_type:         "ELB" => "ELB"
    launch_configuration:      "terraform-20161wu" =>
"${aws_launch_configuration.example.id}"
    max_size:                  "2" => "2"
    min_elb_capacity:          "" => "2"
    min_size:                  "2" => "2"
    name:                      "tf-asg-200170wox" => "${var.cluster_nam
-${aws_launch_configuration.example.name}" (forces new resource)
    protect_from_scale_in:     "false" => "false"
    tag.#:                     "1" => "1"
    tag.2305202985.key:        "Name" => "Name"
    tag.2305202985.value:      "webservers-stage" => "webservers-stage"
    vpc_zone_identifier.#:     "1" => "<computed>"
    wait_for_capacity_timeout: "10m" => "10m"

-/+ module.webserver_cluster.aws_launch_configuration.example
    ebs_block_device.#:  "0" => "<computed>"
    ebs_optimized:       "false" => "<computed>"
    enable_monitoring:   "true" => "true"
    image_id:            "ami-40d28157" => "ami-40d28157"
    instance_type:       "t2.micro" => "t2.micro"
    key_name:            "" => "<computed>"
    name:                "terraform-20161118182404wu" => "<computed>"
    root_block_device.#: "0" => "<computed>"
    security_groups.#:   "1" => "1"
    user_data:           "416115339b" => "3bab6edc" (forces new resourc

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 new resource" next to its name parameter, which means Terraform will replace it with a new Auto Scaling Group running the new version of your code (or new version of your 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).

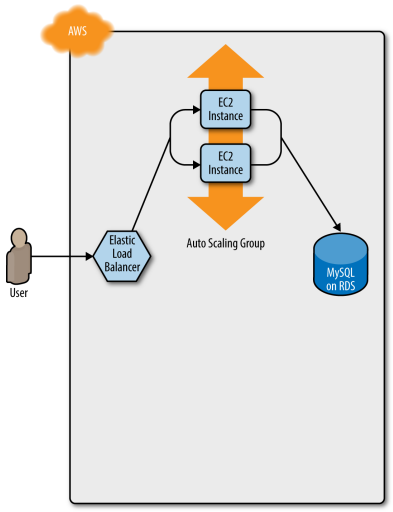

Figure 5-1. Initially, you have the original ASG running v1 of your code

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 start deploying a new ASG with v2 of your code (Figure 5-2).

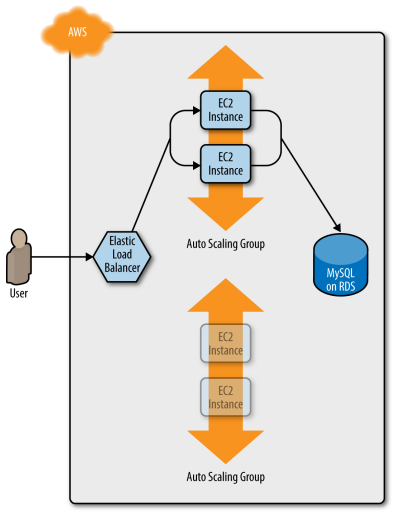

Figure 5-2. Terraform begins deploying the new ASG with v2 of your code

After a minute or two, the servers in the new ASG have booted, connected to the database, and registered in the ELB. 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 ELB happens to route them (Figure 5-3).

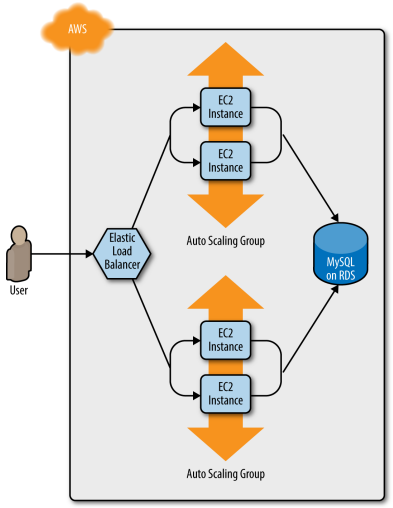

Figure 5-3. The servers in the new ASG boot up, connect to the DB, register in the ELB, and start serving traffic

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

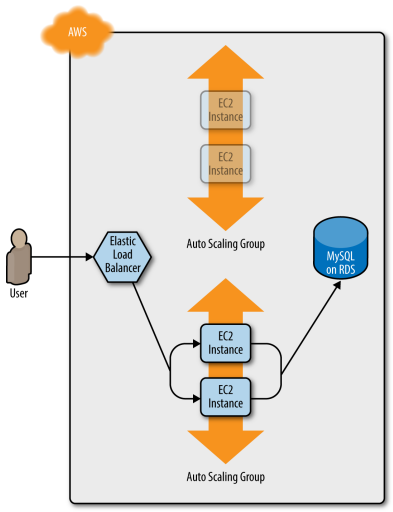

Figure 5-4. The servers in the old ASG begin to shut down

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).

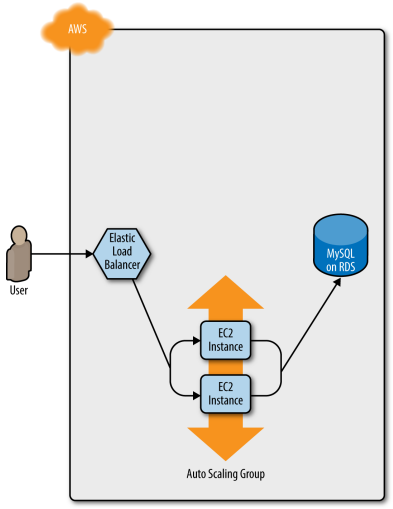

Figure 5-5. Now, only the new ASG remains, which is running v2 of your code

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

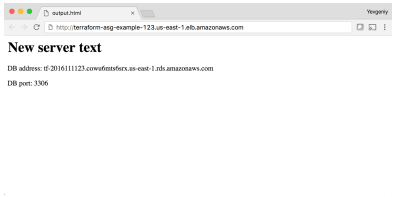

Figure 5-6. The new code is now deployed

Success! The new server text has deployed. As a fun experiment, make another change to the `server_text` parameter (e.g., update it to say "foo bar"), and run the `apply` command. In a separate terminal tab, if you're on Linux/Unix/OS X, you can use a Bash one-liner to run `curl` in a loop, hitting your ELB 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 that says "New server text". Then, you'll start seeing it alternate between the "New server text" and "foo bar". This means the new Instances have registered in the ELB. After another minute, the "New server text" will disappear, and you'll only see "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 was a bug in v2 of your app and it failed to boot, then the Instances in the new ASG will not register with the ELB. 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 ELB, after which it will consider the deployment a failure, delete the v2 ASG, and exit 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 has limitations
- Zero-downtime deployment has limitations
- Valid plans can fail
- Refactoring can be tricky
- Eventual consistency is consistent…eventually

Count Has Limitations

In the examples in this chapter, you made extensive use of the `count` parameter in loops and if-statements. This works well, but there is a significant limitation: you cannot use dynamic data in the count parameter. By "dynamic data," I mean any data that is fetched from a provider (e.g., from a data source) or is only available after a resource has been created (e.g., an output attribute of a resource).

For example, imagine you wanted to deploy multiple EC2 Instances, and for some reason did not want to use an Auto Scaling Group to do it. The code might look something like this:

```
resource "aws_instance" "example" {
  count         = 3
  ami           = "ami-40d28157"
  instance_type = "t2.micro"
}
```

What if you wanted to deploy one EC2 Instance per availability zone (AZ) in the current AWS region? You might be tempted to use the `aws_availability_zones` data source to retrieve the list of AZs and update the code as follows:

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

resource "aws_instance" "example" {
  count             = "${length(data.aws_availability_zones.all.names)}"
  availability_zone =
    "${element(data.aws_availability_zones.all.names, count.index)}"

  ami           = "ami-40d28157"
  instance_type = "t2.micro"
}
```

This code uses the `length` interpolation function to set the `count` parameter to the number of available AZs, and the `element` interpolation function with `count.index` to set the `availability_zone` parameter to a different AZ for each EC2 Instance. This is a perfectly reasonable approach, but unfortunately, if you run this code, you'll get an error that looks like this:

```
aws_instance.example:resource count can't reference resource variable:
data.aws_availability_zones.all.names
```

The cause is that Terraform tries to resolve all the `count` parameters *before* fetching any dynamic data. Therefore, it's trying to parse `${length(data.aws_availability_zones.all.names)}` as a number *before* it has fetched the list of AZs. This is an inherent limitation in Terraform's design and, as of January 2017, it's an open issue in the Terraform community.

For now, your only option is to manually look up how many AZs you have in your AWS region (every AWS account has access to different AZs, so check your EC2 console) and hard-code the `count` parameter to that value:

```
resource "aws_instance" "example" {
  count             = 3
  availability_zone =
    "${element(data.aws_availability_zones.all.names, count.index)}"

  ami           = "ami-40d28157"
  instance_type = "t2.micro"
}
```

Alternatively, you can set the `count` parameter to a variable:

```
resource "aws_instance" "example" {
  count             = "${var.num_availability_zones}"
  availability_zone =
    "${element(data.aws_availability_zones.all.names, count.index)}"

  ami           = "ami-40d28157"
  instance_type = "t2.micro"
}
```

However, the value for that variable must also be hard-coded somewhere along the line (e.g., via a `default` defined with the variable or a value passed in via the command-line `-var` option) and not depend on any dynamic data:

```
variable "num_availability_zones" {
  description = "The number of Availability Zones in the AWS region"
  default     = 3
}
```

Zero-Downtime Deployment has Limitations

Using `create_before_destroy` with an ASG is a great technique for zero-downtime deployment, but there is one limitation: 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 web server 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 would stay that way until 9 a.m. the next day.

There are several possible workarounds, including:

- Change the `recurrence` parameter on the `aws_autoscaling_schedule` from `0 9 * * *`, which means "run at 9 a.m.", to something like `0-59 9-17 * * *`, which means "run every minute from 9 a.m. to 5 p.m." If the ASG already has 10 servers, rerunning this auto scaling policy will have no effect, which is just fine; and if the ASG was just deployed, then running this policy ensures that the ASG won't be around for more than a minute before the number of Instances is increased to 10. This approach is a bit of a hack, and while it may work for scheduled auto scaling actions, it does not work for auto scaling policies triggered by load (e.g., "add two servers if CPU utilization is over 95%").
- Create a custom script that uses the AWS API to figure out how many servers are running in the ASG before deployment, use that value as the `desired_capacity` parameter of the ASG in the Terraform configurations, and then kick off the deployment. After the new ASG has booted, the script should remove the `desired_capacity` parameter so that the auto scaling policies can control the size of the ASG. On the plus side, the replacement ASG will boot up with the same number of servers as the original, and this approach works with all types of auto scaling policies. The downside is that it requires a custom and somewhat complicated deployment script rather than pure Terraform code.

Ideally, Terraform would have first-class support for zero-downtime deployment, but as of January 2017, this is an open issue in the Terraform community.

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 Chapter 2:

```
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:

```
+ aws_iam_user.existing_user
    arn:           "<computed>"
    force_destroy: "false"
    name:          "yevgeniy.brikman"
    path:          "/"
    unique_id:     "<computed>"

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

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

```
Error applying plan:

* aws_iam_user.existing_user: Error creating IAM User yevgeniy.brikman:
EntityAlreadyExists: User with name yevgeniy.brikman already exists.
```

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 it manually or with a different set of Terraform configurations, 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 offguard 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:

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.

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. 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 interpolations, such as `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 for each resource typically specifies how to import it at the bottom of the page.
>
> For example, here is the `import` command 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 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 may want to look into a tool such as Terraforming (http://terraforming.dtan4.net/), 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, the `webserver-cluster` module has an input variable named `cluster_name`:

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

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 uses the `cluster_name` variable in a number of resources, including the `name` parameters of the ELB and two security groups. 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 ELB, there will be nothing to route traffic to your web server cluster until the new ELB 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" {
  name = "${var.cluster_name}-instance"

  lifecycle {
    create_before_destroy = true
  }
}
```

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`. 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:

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.

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 `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.

All identifiers are immutable

> Treat the identifiers you associate with each resource as immutable. If you change an identifier, Terraform will delete the old resource and create a new one to replace it. Therefore, don't rename identifiers unless absolutely necessary, and even then, use the `plan` command, and consider whether you should use a create-before-destroy strategy.

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, Terraform does not do a great job of this. As of version 0.8.x, Terraform still has a number of eventual consistency bugs that you will hit from time to time after running `terraform apply`.

For example, there is #5335:

```
> terraform apply
aws_route.internet-gateway:
error finding matching route for Route table (rtb-5ca64f3b)
and destination CIDR block (0.0.0.0/0)
```

And #5185:

```
> terraform apply
Resource 'aws_eip.nat' does not have attribute 'id' for variable 'aws_e
```

And #6813:

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

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.

It's also worth noting that eventual consistency bugs may be more likely if the place from where you're running Terraform is geographically far away from the provider you're using. For example, if you're running Terraform on your laptop in California and you're deploying code to the AWS region `eu-west-1`, which is thousands of miles away in Ireland, you are more likely to see eventual consistency bugs. I'm guessing this is because the API calls from Terraform get routed to a local AWS data center (e.g., `us-west-1`, which is in California), and the replicas in that data center take a longer time to update if the actual changes are happening in a different data center.

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`, `create_before_destroy`, and interpolation 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 interpolation documentation and let your inner hacker go wild. OK, maybe not too wild, as someone still has to maintain your code, but just wild enough that you can create clean, beautiful APIs for your users.

These users will be the focus of the next chapter, which describes how to use Terraform as a team. This includes a discussion of what workflows you can use, how to manage environments, how to test your Terraform configurations, and more.

---

[1] If the `index` is greater than the number of elements in `list`, the `element` function will automatically "wrap" around using a standard mod function.

[2] You can learn about CPU credits here: *http://amzn.to/2lTuvs5* (http://amzn.to/2lTuvs5).

[3] Credit for this technique goes to Paul Hinze (http://bit.ly/2lksQgv).

Recommended / Queue / History / Topics / Tutorials / Settings / Get the App / Sign Out

PREV
4. How to Create Reusable Infrastructure with Terraform Modules

NEXT
6. How to Use Terraform as a Team