

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