If I browse your *live* repository, I should be able to see, from a quick scan, what resources have been deployed in what environments. That is, every resource should have a 1:1 match with some line of code checked into the *live* repo. This seems obvious at first glance, but it's surprisingly easy to get it wrong. One way to get it wrong, as I just mentioned, is to make out-of-band changes so that the code is there, but the live infrastructure is different. A more subtle way to get it wrong is to use Terraform workspaces to manage environments so that the live infrastructure is there, but the code isn't. That is, if you use workspaces, your *live* repo will have only one copy of the code, even though you may have 3 or 30 environments deployed with it. From merely looking at the code, there will be no way to know what's actually deployed, which will lead to mistakes and make maintenance complicated. Therefore, as described in "Isolation via Workspaces", instead of using workspaces to manage environments, you want each environment defined in a separate folder, using separate files, so that you can see exactly what environments have been deployed just by browsing the *live* repository. Later in this chapter, you'll see how to do this with minimal copying and pasting.

"The main branch…"

You should have to look at only a single branch to understand what's actually deployed in production. Typically, that branch will be `main`. This means that all changes that affect the production environment should go directly into `main` (you can create a separate branch but only to create a pull request with the intention of merging that branch into `main`), and you should run `terraform apply` only for the production environment against the `main` branch. In the next section, I'll explain why.

The trouble with branches

In Chapter 3, you saw that you can use the locking mechanisms built into Terraform backends to ensure that if two team members are running

`terraform apply` at the same time on the same set of Terraform configurations, their changes do not overwrite each other. Unfortunately, this only solves part of the problem. Even though Terraform backends provide locking for Terraform state, they cannot help you with locking at the level of the Terraform code itself. In particular, if two team members are deploying the same code to the same environment but from different branches, you'll run into conflicts that locking can't prevent.

For example, suppose that one of your team members, Anna, makes some changes to the Terraform configurations for an app called "foo" that consists of a single EC2 Instance:

```
resource "aws_instance" "foo" {
  ami           = data.aws_ami.ubuntu.id
  instance_type = "t2.micro"
}
```

The app is getting a lot of traffic, so Anna decides to change the `instance_type` from `t2.micro` to `t2.medium`:

```
resource "aws_instance" "foo" {
  ami           = data.aws_ami.ubuntu.id
  instance_type = "t2.medium"
}
```

Here's what Anna sees when she runs `terraform plan`:

```
$ terraform plan

(...)

Terraform will perform the following actions:

  # aws_instance.foo will be updated in-place
  ~ resource "aws_instance" "foo" {
        ami                          = "ami-0fb653ca2d3203ac1"
        id                           = "i-096430d595c80cb53"
        instance_state               = "running"
      ~ instance_type                = "t2.micro" -> "t2.medium"
        (...)
    }
```

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

Those changes look good, so she deploys them to staging.

In the meantime, Bill comes along and also starts making changes to the Terraform configurations for the same app but on a different branch. All Bill wants to do is to add a tag to the app:

```
resource "aws_instance" "foo" {
  ami           = data.aws_ami.ubuntu.id
  instance_type = "t2.micro"

  tags = {
    Name = "foo"
  }
}
```

Note that Anna's changes are already deployed in staging, but because they are on a different branch, Bill's code still has the `instance_type` set to the old value of `t2.micro`. Here's what Bill sees when he runs the `plan` command (the following log output is truncated for readability):

```
$ terraform plan

(...)

Terraform will perform the following actions:

  # aws_instance.foo will be updated in-place
  ~ resource "aws_instance" "foo" {
        ami                          = "ami-0fb653ca2d3203ac1"
        id                           = "i-096430d595c80cb53"
        instance_state               = "running"
      ~ instance_type                = "t2.medium" -> "t2.micro"
      + tags                         = {
          + "Name" = "foo"
        }
        (...)
    }

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