Uh oh, he's about to undo Anna's `instance_type` change! If Anna is still testing in staging, she'll be very confused when the server suddenly redeploys and starts behaving differently. The good news is that if Bill diligently reads the `plan` output, he can spot the error before it affects Anna. Nevertheless, the point of the example is to highlight what happens when you deploy changes to a shared environment from different branches.

The locking from Terraform backends doesn't help here, because the conflict has nothing to do with concurrent modifications to the state file; Bill and Anna might be applying their changes weeks apart, and the problem would be the same. The underlying cause is that branching and Terraform are a bad combination. Terraform is implicitly a mapping from Terraform code to infrastructure deployed in the real world. Because there's only one real world, it doesn't make much sense to have multiple branches of your Terraform code. So for any shared environment (e.g., stage, prod), always deploy from a single branch.

Run the Code Locally

Now that you've got the code checked out onto your computer, the next step is to run it. The gotcha with Terraform is that, unlike application code, you don't have "localhost"; for example, you can't deploy an AWS ASG onto your own laptop. As discussed in "Manual Testing Basics", the only way to manually test Terraform code is to run it in a sandbox environment, such as an AWS account dedicated for developers (or better yet, one AWS account for each developer).

Once you have a sandbox environment, to test manually, you run `terraform apply`:

```
$ terraform apply

(...)

Apply complete! Resources: 5 added, 0 changed, 0 destroyed.

Outputs:
```