```
path")
    assert_equal(404, status_code)
    assert_equal('text/plain', content_type)
    assert_equal('Not Found', body)
  end
end
```

And here's how you run the tests:

```
$ ruby web-server-test.rb
Loaded suite web-server-test
Finished in 0.000572 seconds.
----------------------------------------
3 tests, 9 assertions, 0 failures, 0 errors
100% passed
----------------------------------------
```

In 0.0005272 seconds, you can now find out whether your web server code works as expected. That's the power of unit testing: a fast feedback loop that helps you build confidence in your code.

Unit testing Terraform code

What is the equivalent of this sort of unit testing with Terraform code? The first step is to identify what a "unit" is in the Terraform world. The closest equivalent to a single function or class in Terraform is a single reusable module such as the `alb` module you created in Chapter 8. How would you test this module?

With Ruby, to write unit tests, you needed to refactor the code so you could run it without complicated dependencies such as `HTTPServer`, `HTTPRequest`, or `HTTPResponse`. If you think about what your Terraform code is doing—making API calls to AWS to create the load balancer, listeners, target groups, and so on—you'll realize that 99% of what this code is doing is communicating with complicated dependencies! There's no practical way to reduce the number of external dependencies to zero, and even if you could, you'd effectively be left with no code to test.[2]

That brings us to *key testing takeaway #3*: you cannot do *pure* unit testing for Terraform code.

But don't despair. You can still build confidence that your Terraform code behaves as expected by writing automated tests that use your code to deploy real infrastructure into a real environment (e.g., into a real AWS account). In other words, unit tests for Terraform are really integration tests. However, I prefer to still call them unit tests to emphasize that the goal is to test a single unit (i.e., a single reusable module) to get feedback as quickly as possible.

This means that the basic strategy for writing unit tests for Terraform is as follows:

1. Create a small, standalone module.
2. Create an easy-to-deploy example for that module.
3. Run `terraform apply` to deploy the example into a real environment.
4. Validate that what you just deployed works as expected. This step is specific to the type of infrastructure you're testing: for example, for an ALB, you'd validate it by sending an HTTP request and checking that you receive back the expected response.
5. Run `terraform destroy` at the end of the test to clean up.

In other words, you do *exactly* the same steps as you would when doing manual testing, but you capture those steps as code. In fact, that's a good mental model for creating automated tests for your Terraform code: ask yourself, "How would I have tested this manually to be confident it works?" and then implement that test in code.

You can use any programming language you want to write the test code. In this book, all of the tests are written in the Go programming language to take advantage of an open source Go library called Terratest, which supports testing a wide variety of infrastructure-as-code tools (e.g., Terraform, Packer, Docker, Helm) across a wide variety of environments (e.g., AWS, Google Cloud, Kubernetes). Terratest is a bit like a Swiss Army knife, with hundreds of tools built in that make it significantly easier to test

infrastructure code, including first-class support for the test strategy just described, where you `terraform apply` some code, validate that it works, and then run `terraform destroy` at the end to clean up.

To use Terratest, you need to do the following:

1. Install Go (minimum version 1.13).
2. Create a folder for your test code: e.g., a folder named *test*.
3. Run `go mod init <NAME>` in the folder you just created, where `NAME` is the name to use for this test suite, typically in the format `github.com/<ORG_NAME>/<PROJECT_NAME>` (e.g., `go mod init github.com/brikis98/terraform-up-and-running`). This should create a *go.mod* file, which is used to track the dependencies of your Go code.

As a quick sanity check that your environment is set up correctly, create *go_sanity_test.go* in your new folder with the following contents:

```
package test

import (
        "fmt"
        "testing"
)

func TestGoIsWorking(t *testing.T) {
        fmt.Println()
        fmt.Println("If you see this text, it's working!")
        fmt.Println()
}
```

Run this test using the `go test` command:

```
go test -v
```

The `-v` flag means verbose, which ensures that the test always shows all log output. You should see output that looks something like this:

```
=== RUN   TestGoIsWorking

If you see this text, it's working!

--- PASS: TestGoIsWorking (0.00s)
PASS
ok      github.com/brikis98/terraform-up-and-running-code
0.192s
```

If that's working, feel free to delete *go_sanity_test.go*, and move on to writing a unit test for the `alb` module. Create *alb_example_test.go* in your *test* folder with the following skeleton of a unit test:

```
package test

import (
        "testing"
)

func TestAlbExample(t *testing.T) {
}
```

The first step is to direct Terratest to where your Terraform code resides by using the `terraform.Options` type:

```
package test

import (
        "github.com/gruntwork-io/terratest/modules/terraform"
        "testing"
)

func TestAlbExample(t *testing.T) {
        opts := &terraform.Options{
                // You should update this relative path to point
at your alb
                // example directory!
                TerraformDir: "../examples/alb",
        }
}
```

Note that to test the `alb` module, you actually test the example code in your *examples* folder (you should update the relative path in

`TerraformDir` to point to the folder where you created that example).

The next step in the automated test is to run `terraform init` and `terraform apply` to deploy the code. Terratest has handy helpers for doing that:

```
func TestAlbExample(t *testing.T) {
        opts := &terraform.Options{
                // You should update this relative path to point
at your alb
                // example directory!
                TerraformDir: "../examples/alb",
        }

        terraform.Init(t, opts)
        terraform.Apply(t, opts)
}
```

In fact, running `init` and `apply` is such a common operation with Terratest that there is a convenient `InitAndApply` helper method that does both in one command:

```
func TestAlbExample(t *testing.T) {
        opts := &terraform.Options{
                // You should update this relative path to point
at your alb
                // example directory!
                TerraformDir: "../examples/alb",
        }

        // Deploy the example
        terraform.InitAndApply(t, opts)
}
```

The preceding code is already a fairly useful unit test, since it will run `terraform init` and `terraform apply` and fail the test if those commands don't complete successfully (e.g., due to a problem with your Terraform code). However, you can go even further by making HTTP requests to the deployed load balancer and checking that it returns the data you expect. To do that, you need a way to get the domain name of the

deployed load balancer. Fortunately, that's available as an output variable in the `alb` example:

```
output "alb_dns_name" {
  value       = module.alb.alb_dns_name
  description = "The domain name of the load balancer"
}
```

Terratest has helpers built in to read outputs from your Terraform code:

```
func TestAlbExample(t *testing.T) {
        opts := &terraform.Options{
                // You should update this relative path to point
at your alb
                // example directory!
                TerraformDir: "../examples/alb",
        }

        // Deploy the example
        terraform.InitAndApply(t, opts)

        // Get the URL of the ALB
        albDnsName := terraform.OutputRequired(t, opts,
"alb_dns_name")
        url := fmt.Sprintf("http://%s", albDnsName)
}
```

The `OutputRequired` function returns the output of the given name, or it fails the test if that output doesn't exist or is empty. The preceding code builds a URL from this output using the `fmt.Sprintf` function that's built into Go (don't forget to import the `fmt` package). The next step is to make some HTTP requests to this URL using the `http_helper` package (make sure to add `github.com/gruntwork-io/terratest/modules/http-helper` as an import):

```
func TestAlbExample(t *testing.T) {
        opts := &terraform.Options{
                // You should update this relative path to point
at your alb
                // example directory!
                TerraformDir: "../examples/alb",
```

```
        }

        // Deploy the example
        terraform.InitAndApply(t, opts)

        // Get the URL of the ALB
        albDnsName := terraform.OutputRequired(t, opts,
"alb_dns_name")
        url := fmt.Sprintf("http://%s", albDnsName)

        // Test that the ALB's default action is working and
returns a 404
        expectedStatus := 404
        expectedBody := "404: page not found"
        maxRetries := 10
        timeBetweenRetries := 10 * time.Second

        http_helper.HttpGetWithRetry(
                t,
                url,
                nil,
                expectedStatus,
                expectedBody,
                maxRetries,
                timeBetweenRetries,
        )
}
```

The `http_helper.HttpGetWithRetry` method will make an HTTP GET request to the URL you pass in and check that the response has the expected status code and body. If it doesn't, the method will retry up to the specified maximum number of retries, with the specified amount of time between retries. If it eventually achieves the expected response, the test will pass; if the maximum number of retries is reached without the expected response, the test will fail. This sort of retry logic is very common in infrastructure testing, as there is usually a period of time between when `terraform apply` finishes and when the deployed infrastructure is completely ready (i.e., it takes time for health checks to pass, DNS updates to propagate, and so on), and as you don't know exactly how long that'll take, the best option is to retry until it works or you hit a timeout.

The last thing you need to do is to run `terraform destroy` at the end of the test to clean up. As you can guess, there is a Terratest helper for this: `terraform.Destroy`. However, if you call `terraform.Destroy` at the very end of the test, if any of the code before that causes a test failure (e.g., `HttpGetWithRetry` fails because the ALB is misconfigured), the test code will exit before getting to `terraform.Destroy`, and the infrastructure deployed for the test will never be cleaned up.

Therefore, you want to ensure that you *always* run `terraform.Destroy`, even if the test fails. In many programming languages, this is done with a `try` / `finally` or `try` / `ensure` construct, but in Go, this is done by using the `defer` statement, which will guarantee that the code you pass to it will be executed when the surrounding function returns (no matter how that return happens):

```
func TestAlbExample(t *testing.T) {
        opts := &terraform.Options{
                // You should update this relative path to point
at your alb
                // example directory!
                TerraformDir: "../examples/alb",
        }

        // Clean up everything at the end of the test
        defer terraform.Destroy(t, opts)

        // Deploy the example
        terraform.InitAndApply(t, opts)

        // Get the URL of the ALB
        albDnsName := terraform.OutputRequired(t, opts,
"alb_dns_name")
        url := fmt.Sprintf("http://%s", albDnsName)

        // Test that the ALB's default action is working and
returns a 404
        expectedStatus := 404
        expectedBody := "404: page not found"
        maxRetries := 10
        timeBetweenRetries := 10 * time.Second

        http_helper.HttpGetWithRetry(
```

```
                t,
                url,
                nil,
                expectedStatus,
                expectedBody,
                maxRetries,
                timeBetweenRetries,
            )
    }
```

Note that the `defer` is added early in the code, even before the call to `terraform.InitAndApply`, to ensure that nothing can cause the test to fail before getting to the `defer` statement and preventing it from queueing up the call to `terraform.Destroy`.

OK, this unit test is finally ready to run!

> **TERRATEST VERSION**
>
> The test code in this book was written with Terratest v0.39.0. Terratest is still a pre-1.0.0 tool, so newer releases may contain backward-incompatible changes. To ensure the test examples in this book work as written, I recommend installing Terratest specifically at version v0.39.0, and not the latest version. To do that, go into *go.mod* and add the following to the end of the file:
>
> ```
> require github.com/gruntwork-io/terratest v0.39.0
> ```

Since this is a brand-new Go project, as a one-time action, you need to tell Go to download dependencies (including Terratest). The easiest way to do that at this stage is to run the following:

```
go mod tidy
```

This will download all your dependencies and create a *go.sum* file to lock the exact versions you used.

Next, since this test deploys infrastructure to AWS, before running the test, you need to authenticate to your AWS account as usual (see "Other AWS

Authentication Options"). You saw earlier in this chapter that you should do manual testing in a sandbox account; for automated testing, this is even more important, so I recommend authenticating to a totally separate account. As your automated test suite grows, you might be spinning up hundreds or thousands of resources in every test suite, so keeping them isolated from everything else is essential.

I typically recommend that teams have a completely separate environment (e.g., completely separate AWS account) just for automated testing—separate even from the sandbox environments you use for manual testing. That way, you can safely delete all resources that are more than a few hours old in the testing environment, based on the assumption that no test will run that long.

After you've authenticated to an AWS account that you can safely use for testing, you can run the test, as follows:

```
$ go test -v -timeout 30m

TestAlbExample 2019-05-26T13:29:32+01:00 command.go:53:
Running command terraform with args [init -upgrade=false]

(...)

TestAlbExample 2019-05-26T13:29:33+01:00 command.go:53:
Running command terraform with args [apply -input=false -
lock=false]

(...)

TestAlbExample 2019-05-26T13:32:06+01:00 command.go:121:
Apply complete! Resources: 5 added, 0 changed, 0 destroyed.

(...)

TestAlbExample 2019-05-26T13:32:06+01:00 command.go:53:
Running command terraform with args [output -no-color
alb_dns_name]

(...)

TestAlbExample 2019-05-26T13:38:32+01:00 http_helper.go:27:
Making an HTTP GET call to URL
```

```
http://terraform-up-and-running-1892693519.us-east-
2.elb.amazonaws.com

(...)

TestAlbExample 2019-05-26T13:38:32+01:00 command.go:53:
Running command terraform with args
[destroy -auto-approve -input=false -lock=false]

(...)

TestAlbExample 2019-05-26T13:39:16+01:00 command.go:121:
Destroy complete! Resources: 5 destroyed.

(...)

PASS
ok      terraform-up-and-running        229.492s
```

Note the use of the `-timeout 30m` argument with `go test`. By default, Go imposes a time limit of 10 minutes for tests, after which it forcibly kills the test run, not only causing the tests to fail but also preventing the cleanup code (i.e., `terraform destroy`) from running. This ALB test should take closer to five minutes, but whenever running a Go test that deploys real infrastructure, it's safer to set an extra-long timeout to avoid the test being killed partway through and leaving all sorts of infrastructure still running.

The test will produce a lot of log output, but if you read through it carefully, you should be able to spot all of the key stages of the test:

1. Running `terraform init`
2. Running `terraform apply`
3. Reading output variables using `terraform output`
4. Repeatedly making HTTP requests to the ALB
5. Running `terraform destroy`

It's nowhere near as fast as the Ruby unit tests, but in less than five minutes, you can now automatically find out whether your `alb` module works as