exactly, as the User Data script in the `hello-world-app` module returns an HTML response with other text in it as well.

Alright, run the integration test to see whether it worked:

```
$ go test -v -timeout 30m -run "TestHelloWorldAppStage"

(...)

PASS
ok      terraform-up-and-running        795.63s
```

Excellent, you now have an integration test that you can use to check that several of your modules work correctly together. This integration test is more complicated than the unit test, and it takes more than twice as long (10–15 minutes rather than 4–5 minutes). In general, there's not much that you can do to make things *faster*—the bottleneck here is how long AWS takes to deploy and undeploy RDS, ASGs, ALBs, etc.—but in certain circumstances, you might be able to make the test code do *less* using *test stages*.

Test stages

If you look at the code for your integration test, you may notice that it consists of five distinct "stages":

1. Run `terraform apply` on the `mysql` module.
2. Run `terraform apply` on the `hello-world-app` module.
3. Run validations to make sure everything is working.
4. Run `terraform destroy` on the `hello-world-app` module.
5. Run `terraform destroy` on the `mysql` module.

When you run these tests in a CI environment, you'll want to run all of the stages, from start to finish. However, if you're running these tests in your local dev environment while iteratively making changes to the code, running all of these stages is unnecessary. For example, if you're making

changes only to the `hello-world-app` module, rerunning this entire test after every change means you're paying the price of deploying and undeploying the `mysql` module, even though none of your changes affect it. That adds 5 to 10 minutes of pure overhead to every test run.

Ideally, the workflow would look more like this:

1. Run `terraform apply` on the `mysql` module.
2. Run `terraform apply` on the `hello-world-app` module.
3. Now, you start doing iterative development:
    a. Make a change to the `hello-world-app` module.
    b. Rerun `terraform apply` on the `hello-world-app` module to deploy your updates.
    c. Run validations to make sure everything is working.
    d. If everything works, move on to the next step. If not, go back to step 3a.
4. Run `terraform destroy` on the `hello-world-app` module.
5. Run `terraform destroy` on the `mysql` module.

Having the ability to quickly do that inner loop in step 3 is the key to fast, iterative development with Terraform. To support this, you need to break your test code into *stages*, in which you can choose the stages to execute and those that you can skip.

Terratest supports this natively with the `test_structure` package. The idea is that you wrap each stage of your test in a function with a name, and you can then direct Terratest to skip some of those names by setting environment variables. Each test stage stores test data on disk so that it can be read back from disk on subsequent test runs. Let's try this out on *test/hello_world_integration_test.go*, writing the skeleton of the test first and then filling in the underlying methods later:

```
func TestHelloWorldAppStageWithStages(t *testing.T) {
        t.Parallel()

        // Store the function in a short variable name solely to
make the
        // code examples fit better in the book.
        stage := test_structure.RunTestStage

        // Deploy the MySQL DB
        defer stage(t, "teardown_db", func() { teardownDb(t,
dbDirStage) })
        stage(t, "deploy_db", func() { deployDb(t, dbDirStage) })

        // Deploy the hello-world-app
        defer stage(t, "teardown_app", func() { teardownApp(t,
appDirStage) })
        stage(t, "deploy_app", func() { deployApp(t, dbDirStage,
appDirStage) })

        // Validate the hello-world-app works
        stage(t, "validate_app", func() { validateApp(t,
appDirStage) })
}
```

The structure is the same as before—deploy `mysql`, deploy `hello-world-app`, validate `hello-world-app`, undeploy `hello-world-app` (runs at the end due to `defer`), undeploy `mysql` (runs at the end due to `defer`)—except now, each stage is wrapped in `test_structure.RunTestStage`. The `RunTestStage` method takes three arguments:

`t`

: The first argument is the `t` value that Go passes as an argument to every automated test. You can use this value to manage test state. For example, you can fail the test by calling `t.Fail()`.

Stage name

: The second argument allows you to specify the name for this test stage. You'll see an example shortly of how to use this name to skip test stages.

The code to execute

The third argument is the code to execute for this test stage. This can be any function.

Let's now implement the functions for each test stage, starting with `deployDb`:

```
func deployDb(t *testing.T, dbDir string) {
        dbOpts := createDbOpts(t, dbDir)

        // Save data to disk so that other test stages executed
at a later
        // time can read the data back in
        test_structure.SaveTerraformOptions(t, dbDir, dbOpts)

        terraform.InitAndApply(t, dbOpts)
}
```

Just as before, to deploy `mysql`, the code calls `createDbOpts` and `terraform.InitAndApply`. The only new thing is that, in between those two steps, there is a call to `test_structure.SaveTerraformOptions`. This writes the data in `dbOpts` to disk so that other test stages can read it later on. For example, here's the implementation of the `teardownDb` function:

```
func teardownDb(t *testing.T, dbDir string) {
        dbOpts := test_structure.LoadTerraformOptions(t, dbDir)
        defer terraform.Destroy(t, dbOpts)
}
```

This function uses `test_structure.LoadTerraformOptions` to load the `dbOpts` data from disk that was earlier saved by the `deployDb` function. The reason you need to pass this data via the hard drive rather than passing it in memory is that you can run each test stage as part of a different test run—and therefore, as part of a different process. As you'll see a little later in this chapter, on the first few runs of `go test`, you might want to run `deployDb` but skip `teardownDb`, and then in later

runs do the opposite, running `teardownDb` but skipping `deployDb`. To ensure that you're using the same database across all those test runs, you must store that database's information on disk.

Let's now implement the `deployHelloApp` function:

```
func deployApp(t *testing.T, dbDir string, helloAppDir string) {
        dbOpts := test_structure.LoadTerraformOptions(t, dbDir)
        helloOpts := createHelloOpts(dbOpts, helloAppDir)

        // Save data to disk so that other test stages executed
at a later
        // time can read the data back in
        test_structure.SaveTerraformOptions(t, helloAppDir,
helloOpts)

        terraform.InitAndApply(t, helloOpts)
}
```

This function reuses the `createHelloOpts` function from before and calls `terraform.InitAndApply` on it. Again, the only new behavior is the use of `test_structure.LoadTerraformOptions` to load `dbOpts` from disk and the use of `test_structure.SaveTerraformOptions` to save `helloOpts` to disk. At this point, you can probably guess what the implementation of the `teardownApp` method looks like:

```
func teardownApp(t *testing.T, helloAppDir string) {
        helloOpts := test_structure.LoadTerraformOptions(t,
helloAppDir)
        defer terraform.Destroy(t, helloOpts)
}
```

And the implementation of the `validateApp` method:

```
func validateApp(t *testing.T, helloAppDir string) {
        helloOpts := test_structure.LoadTerraformOptions(t,
helloAppDir)
        validateHelloApp(t, helloOpts)
}
```

So, overall, the test code is identical to the original integration test, except each stage is wrapped in a call to `test_structure.RunTestStage`, and you need to do a little work to save and load data to and from disk. These simple changes unlock an important ability: you can instruct Terratest to skip any test stage called `foo` by setting the environment variable `SKIP_foo=true`. Let's go through a typical coding workflow to see how this works.

Your first step will be to run the test but to skip both of the teardown stages so that the `mysql` and `hello-world-app` modules stay deployed at the end of the test. Because the teardown stages are called `teardown_db` and `teardown_app`, you need to set the `SKIP_teardown_db` and `SKIP_teardown_app` environment variables, respectively, to direct Terratest to skip those two stages:

```
$ SKIP_teardown_db=true \
  SKIP_teardown_app=true \
  go test -timeout 30m -run 'TestHelloWorldAppStageWithStages'

(...)

The 'SKIP_deploy_db' environment variable is not set,
so executing stage 'deploy_db'.

(...)

The 'deploy_app' environment variable is not set,
so executing stage 'deploy_db'.

(...)

The 'validate_app' environment variable is not set,
so executing stage 'deploy_db'.

(...)

The 'teardown_app' environment variable is set,
so skipping stage 'deploy_db'.

(...)

The 'teardown_db' environment variable is set,
```

```
so skipping stage 'deploy_db'.

(...)

PASS
ok      terraform-up-and-running        423.650s
```

Now you can start iterating on the `hello-world-app` module, and each time you make a change, you can rerun the tests, but this time, direct them to skip not only the teardown stages but also the `mysql` module deploy stage (because `mysql` is still running) so that the only things that execute are `deploy app` and the validations for the `hello-world-app` module:

```
$ SKIP_teardown_db=true \
  SKIP_teardown_app=true \
  SKIP_deploy_db=true \
  go test -timeout 30m -run 'TestHelloWorldAppStageWithStages'

(...)

The 'SKIP_deploy_db' environment variable is set,
so skipping stage 'deploy_db'.

(...)

The 'deploy_app' environment variable is not set,
so executing stage 'deploy_db'.

(...)

The 'validate_app' environment variable is not set,
so executing stage 'deploy_db'.

(...)

The 'teardown_app' environment variable is set,
so skipping stage 'deploy_db'.

(...)

The 'teardown_db' environment variable is set,
so skipping stage 'deploy_db'.
```

```
(...)

PASS
ok      terraform-up-and-running        13.824s
```

Notice how fast each of these test runs is now: instead of waiting 10 to 15 minutes after every change, you can try out new changes in 10 to 60 seconds (depending on the change). Given that you're likely to rerun these stages dozens or even hundreds of times during development, the time savings can be massive.

Once the `hello-world-app` module changes are working the way you expect, it's time to clean everything up. Run the tests once more, this time skipping the deploy and validation stages so that only the teardown stages are executed:

```
$ SKIP_deploy_db=true \
  SKIP_deploy_app=true \
  SKIP_validate_app=true \
  go test -timeout 30m -run 'TestHelloWorldAppStageWithStages'

(...)

The 'SKIP_deploy_db' environment variable is set,
so skipping stage 'deploy_db'.

(...)

The 'SKIP_deploy_app' environment variable is set,
so skipping stage 'deploy_app'.

(...)

The 'SKIP_validate_app' environment variable is set,
so skipping stage 'validate_app'.

(...)

The 'SKIP_teardown_app' environment variable is not set,
so executing stage 'teardown_app'.

(...)

The 'SKIP_teardown_db' environment variable is not set,
```